ANSI C++

ANSI C++

객체지향 쏘프트웨어

개념

기본사상

UML

평양콤퓨터기술대학
외국문도서출판사
주체91

ANSI C++
객체지향쏘프트웨어

평양콤퓨터기술대학
외국문도서출판사

차 례

머리말

1 프로그람작성에 대한 소개

2 쏘프트웨어설계

3 C++에 대한 소개(1)

4 C++에 대한 소개(2)

5 클라스

6 함수

7 분할콤파일

8 배렬

9 본보기

10 정적변수와 함수

11 계승

16 다형성

17 지적자와 동적기억기

18 다형성의 재고찰

19 선언과 강제형변환

20 용기클라스

21 전처리지령

25 STL 용기

26 C++유산콤파일러의 리용

27 속성

머 리 말

이 책은 객체지향언어 ANSI C++를 서술한 책으로서 C언어는 물론 프로그람작성에 대한 기초지식이 없이도 읽을수 있게 되여 있다.

1장에서는 프로그람작성의 객체지향적관점에 대하여 설명하고 2, 3장에서는 C++언어의 기본자료구조와 조종구조를 설명한다. 그리고 교갑화, 계승, 다형성의 개념들을 서술하면서 언어의 객체지향성이란 무엇인가를 설명한다. 이러한 개념들을 서술하는데서 초기의 론의는 C++언어의 고수준기능에 국한되여 있었는데 사실 string클라스와 같은 구조들을 리용함으로써 C++언어의 저수준기능을 리용해야 할 필요는 없었다.

12장에서는 이 책의 앞부분에서 제기한 개념들과 수법들을 실례로 보여 주기 위하여 한가지 유희문제를 설계하고 실현하는 과정을 서술한다. 이 장에서 독자들은 문제의 서술이라는 초기단계로부터 설계과정을 거쳐 유희프로그람을 실제적으로 실현하는 수준에 이르게 될것이다.

책에서는 프로그람구조에서의 이러한 기본적인 문제점들을 고찰한 다음 C++언어의 저수준기능들 특히 지적자, 주소계산과 관련되는 저준위기능들을 서술한다. 이러한 지적자와 주소계산에 대한 서술은 C++언어사용자들이 고수준구조를 실현하는 클라스를 작성할 때에만 리용할수 있도록 하기 위하여 마지막장에까지 계속 진행하였다.

범용알고리듬과 표준용기에 대한 장들에서 표준본보기서고(STL)에 대하여 서술하였다.

또한 C++프로그람의 속성들과 이 언어의 중요구조들을 요약하여 서술한 장들도 있다. 매장의 마감에는 독자들을 위하여 자체평가문제와 련습문제들을 주었다.

이 책의 실례프로그람들에서 쓰이는 항목들은 다음의 약속에 따른다.

프로그람에서 항목	실 례	약 속
실제파라메터	value = 2; display (value);	소문자이다.
클라스	**class** Account { };	클라스이름의 첫 문자는 대문자로 시작한다.
클라스속성/클라스성원변수: (클라스의 모든 구체레들에서 공유되는 대역적인 자료항목)	**static float** the_rate;	the_로 시작하는 소문자이며 클라스의 비공개부에서 선언된다.
상수	**const** MAX = 10;	대문자이다.
렬거형	**enum** Colour = {RED, BLUE};	대문자로 시작한다.
렬거값	**enum** Colour = {RED, BLUE};	대문자이다.
형식파라메터	display (int amount)	소문자이다.
함수	process ( );	소문자이다.
함수적응자	bind1st	소문자이다.
함수객체	less<int> ( )	소문자이다.
구체레속성/클라스성원변수 (객체안에 포함된 자료항목)	**float** the_balance;	the_로 시작하며 소문자이다.
구체레메쏘드/성원함수	picture**.**display ( )	함수이름은 소문자이다.
마크로이름	NAME	대문자이다.
사용자정의형	**typedef** char* C_string;	대문자로 시작한다.
변수이름/객체	mine p_ch	소문자이다. 지적자를 가지는 변수는 p_로 시작한다.

이 책에서 리용되고 있는 기본용어들은 다음과 같다.

검토자(inspector)

객체의 상태를 변화시키지 않는 메쏘드.

교갑화(encapsulation)

공개대면부를 밀착된 함수를 제공하는 자료수속들과 함수들의 은폐된(비공개) 모임으로 만드는것.

구체례메쏘드(instance method)

객체에 들어 있는 구체례속성(성원자료항목)에 접근하는 클라스안의 함수. 실례로 메쏘드 account_balance 는 구체례속성 the_balance 를 호출한다.

```
float  Account::account_balance ()
{
   return  the_balance;
}
```

구체례속성(instance attribute)

객체에 들어 있는 자료요소. C++에서는 이것을 클라스의 자료성원이라고 한다. 클라스의 자료성원들은 클라스명세부의 비공개부에서 선언된다.

```
class  Account
{
  private:
     float   the_balance;
}
```

구체례제시(instantiation)

지정된 형의 항목을 처리하는 객체를 만드는것. 실례로

```
Vector <int> numbers;
```

에서 numbers 는 Vector <int> 클라스의 구체례제시이다.

기초클라스(base class)

다른 클라스들을 파생시키는 클라스. 다른 언어에서는 상위클라스(superclass)라고 한다.

객체(object)

클라스의 구체례. 객체는 클라스의 메쏘드에 의하여 접근, 변화되는 상태를 가진다. 클라스Account의 구체례인 객체 mike는 다음과 같이 선언된다.

```
Account mike;
```

계승(inheritance)

이미 정의된 클라스(기초클라스)로부터 다른(새로운) 클라스(파생클라스)의 파생. 파생클라스는 자기의 메쏘드와 구체례/클라스속성과 함께 기초클라스에서 정의된 메쏘드와 구체례/클라스속성을 가진다. 실례로 클라스 Account에서 파생되는 클라스 Account_with_statement는 다음과 같이 서술된다.

```
class  Account_with_statement : public Account
{
public:
    Account_with_statement ( const  std::string =  “ ”  );
    void  statement ( ostream& );
private:
    std::string  the_account_name;
    int          the_statement_no;
}
```

다중계승(multiple inheritance)

둘이상의 기초클라스로부터 파생된 클라스.

다중정의(overloading)

한개의 식별자나 연산자가 여러가지 의미를 가지는 경우. 실례로 추출연산자 << 는 출력연산자의 의미를 가지고 다중정의될수 있다.

```
std::cout << “ The sum of 1+2 is ” << 1+2 << ‘\n’;
```

다형성(polymorhism)

콤파일시에 그 형이 알려 지지 않는 객체에로 통보문을 보낼수 있는 능력. 메쏘드의 선택은 객체형에 따른다. 실례로 통보문 ‘display’를 이종집합 picture_elements에 들어 있는 각이한 류형의 그림요소들이 받는다.

```
picture_elements[ i ] -> display ( );
```

동적맺기(dynamic binding)

객체와 그 객체에 보내지는 통보문의 맺기가 콤파일시에 알려 지지 않는 경우.

메타클라스(meta-class)
메타클라스의 구체례는 클라스이다. C++에서는 메타클라스를 제공하지 않는다.

메쏘드(method)
객체의 거동을 실현한다. 메쏘드는 클라스에서 함수로써 실현된다. 메쏘드는 클라스메쏘드가 될수도 있고 구체례메쏘드가 될수도 있다.

변이자(mutator)
객체의 상태를 변화시키는 메쏘드.

본보기클라스(templated class)
클라스본체에서 사용되는 한개이상의 형들로 파라메터화된 클라스. 실례로 본보기클라스 vector 의 구체례인 객체 colours 의 선언은 다음과 같다.

```
std::vector < std:string > colours ;
```

실제파라메터(actual parameter)
함수에 넘겨 지는 물리적인 항목. 다음의 실례프로그람토막에서 함수 print에 넘겨 지는 실제파라메터는 int number이다.

```
int  number = 2;

print ( number );
```

정적맺기(static binding)
객체의 메쏘드와 그 객체에 보내지는 통보문의 맺기가 콤파일시에 알려 지는 경우.

클라스(class)
어떤 형과 그 형의 구체례에서 수행되는 연산들이 서술된것. 하나의 클라스는 공통적인 구조와 거동을 공유하는 객체들을 만들어 내는데 쓰인다. 클라스 Account 의 명세부는 다음과 같다.

```
class Account {
public:
    Account ( );
    float  account_balance ( );
    float  withdraw ( float );
    void  deposit ( float );
    void  set_min_balance ( float );
private:
```

```
    float  the_balance;
    float  the_min_balance;
};
```

클라스문자렬(class string)

클라스 std::string 의 구체례에 들어 있는 문자렬. 이 문자렬에 대하여 값주기와 비교를 할수 있으며 그 결과들은 모순되지 않는다.

클라스메쏘드(class method)

클라스속성들에만 접근하는 클라스의 성원함수. 실례로 클라스속성 the_no_transactions 를 설정하는 클라스 Account_R 의 메쏘드 prelude 는 다음과 같다.

```
void Account_R::prelude ( )
{
      the_no_transactions = 0;
}
```

주의: prelude 는 클라스메쏘드이므로 그 클라스의 구체례에 관계없이 성원함수가 호출된다. 실례로 Account_R::prelude();

클라스속성(class attribute)

클라스의 모든 객체들이 공유하는 자료요소. 클라스속성은 실제에 있어서 클라스에서 메쏘드들에 의해서만 접근될수 있는 대역변수이다. 클라스속성은 클라스의 비공개부에서 선언된다. 실례로 클라스 Account_R 의 클라스속성 the_no_transactions 은 다음과 같다.

```
class  Account_R {
   …
private:
     float  the_balance;
     float  the_overdraft;
     static int  the_no_transactions;
};
```

통보문(message)

객체에서 연산되는 메쏘드에 자료값들을 보내는것. 실례로 C++에서 통보문 《구좌 mike 에 30 파운드를 저금한다.》는 다음과 같다.

```
mike.deposit( 30.00 );
```

형식파라메터(formal parameter)

함수본체에서 함수에 넘겨 지는 항목의 이름. 실례로 다음의 코드토막에서 함수 print 의 형식파라메터는 value 이다.

```
void print ( int value )
{
    std::cout << value;
}
```

ADT(Abstract Data Type)

추상자료형. 자료형은 다음과 같은 두개의 부분으로 되여 있다.

- 추상자료형의 구체례에서 쓰이는 공개부안의 연산들.
- 추상자료형에서 비공개부안의 물리적실현부.

(추상자료형의 구체례에 대한 표시와 그 구체례에서 쓰이는 연산들의 실현부로 되여 있다.)

C

데니스 리치에(Dennis Ritchie)에 의하여 처음으로 설계된 언어로서 UNIX조작체계를 작성하는데 쓰이였다. C++는 거의 이 언어에 기초하고 있다. C언어는 B언어와 BCPL언어에 기초하여 설계되였다.

C++문자렬(C++ string)

기억기에 들어 있는 문자들의 배렬. 문자 '\0' 으로 끝난다. 문자렬형은 char * 이다.

1 프로그람작성에 대한 소개

프로그람작성자는 콤퓨터체계가 리해할수 있는 명령들로 문제풀이를 표현하기 위해 프로그람작성언어를 리용한다. 이 장에서는 간단한 문제를 실례로 하여 콤퓨터 프로그람작성언어 C++의 리용법에 대하여 서술한다.

1.1 콤퓨터프로그람작성

콤퓨터프로그람작성언어를 리용하여 문제를 풀기 위해서는 세심한 공정을 거쳐야 한다. 본질에 있어서 그러한 문제는 모든 세부가 정확하면서도 문법적으로 째인 언어에 의하여 표현된다. 그러나 이러한 공정들이 아무리 복잡하다고 할지라도 프로그람작성에 의한 문제풀이는 그것이 가져다 주는 막대한 리득에 비추어 볼 때 해볼만한 가치가 있는것이다.

우리가 살고 있는 세계에 많은 민족어가 있듯이 콤퓨터세계에도 여러 종류의 프로그람작성언어가 있다. 여기서 프로그람작성언어 C++는 오늘날 널리 리용되고 있는 프로그람작성언어들중의 하나이다.

1.2 프로그람작성언어

콤퓨터를 쓰던 1950년대 초기에는 콤퓨터의 기계어로 프로그람을 작성해야 하였다. 그후 인차 프로그람작성자들은 기계어를 기호적으로 표시할수 있는 아쎔블리언어에 의하여 프로그람을 작성할수 있게 되였다. 이 경우에는 아쎔블러가 프로그람작성자가 쓴 기호어명령들을 콤퓨터의 실제적인 기계코드명령으로 콤파일한다. 실례로 아쎔블리언어를 리용하여 일정한 량의 사과값을 계산하는 프로그람은 다음과 같다.

```
LDA     AMOUNT_OF_OF_APPLES        ; 축적기 #에 무게(폰드)를 넣기
MLT     PRICE_PER_POUND            ; 사과의 폰드당 값으로 곱하기
STA     COST_OF_APPLES             ; 결과 보관
```

주의: 매개의 아쎔블리언어명령은 기계코드명령에 대응된다.

1957 - 1958년사이에 고수준언어들인 FORTRAN과 COBOL의 첫판이 나왔다. 이 고수준프로그람작성언어들은 장치구조에 의존하지 않으므로 프로그람작성자들은 보다 쉽고 능률적으로 프로그람들을 작성할수 있었다. 이 경우에는 해당한 언어에 대한 콤파일러가 프로그람작성자가 쓴 고수준언어명령들을 그 콤퓨터의 기계어명령들로 콤파일한다. 이 고수준언어의 우점은 다음과 같다.

- 프로그람작성자가 장치에 의존하지 않고 프로그람을 작성할수 있으므로 생산성을 높인다.

• 프로그람이 정확히 작성되기만 하였으면 그것은 서로 다른 콤퓨터들에 해당 기계명령으로 콤파일될수 있으므로 수정하지 않고도 여러 콤퓨터들에서 실행시킬수 있다.

실례로 앞에서 서술한 사과값을 계산하는 프로그람을 FORTRAN으로 서술하면 다음과 같다.

```
COST = PRICE * AMOUNT
```

1.3 프로그람작성언어의 부류

콤퓨터프로그람작성언어가 쓰인 초시기부터 현재까지 고수준언어들의 개수와 부류는 매우 다양해 졌다. 그러나 대부분언어들은 사용자들이 쓰지 않아 사실상 사멸되였다. 프로그람작성언어들중에서 현재 널리 쓰이는 언어들을 형태별로 간단히 분류해 보면 다음의 표와 같다.

언어형태	언어의 주요특징	실례
함수형	문제는 개별적인 함수들로 갈라 진다. 함수는 읽기만 하는 자료값들을 넘겨 받아 새로운 값을 만들어 낸다. 함수자체가 어떤 함수에 파라메터로 넘겨 질수도 있다. 함수에 보내지는 자료는 변화되지 않으므로 개별적인 함수들은 자기의 입력자료를 받자마자 동시에 실행될수 있다.	ML
수속형	문제는 개별적인 수속들이나 부분루틴들로 갈라 진다. 이 가르기는 일반적으로 우로부터 아래로 해나간다. 이 방법에서는 문제의 어떤 부분이 수속으로 될수 있다면 이것을 또다시 개별적인 수속들로 가른다. 그러나 자료는 보통 가르지 않는다.	C Pascal
론리형	문제는 그 고찰방식에 대하여 주종속관계를 규정하는 규칙들로 갈라 진다.	Prolog
객체지향형	문제는 호상작용하는 객체들로 갈라 진다. 매개 객체는 그 객체의 은페된 상태를 조종하는 메쏘드들을 교갑화하고 은페시킨다. 객체에 보내 진 통보문은 이 교갑화된 메쏘드를 불러 내며 그때 그 메쏘드가 요구한 과제를 수행한다.	Ada 95 Eiffel Java Smalltalk

1.3.1 콤퓨터프로그람작성언어

콤퓨터프로그람작성언어(computer programming language)는 문제풀이를 고수준서술로 표현할수 있는 특수한 언어이다. 그러나 자연언어와는 달리 문제풀이에

대한 서술에서 애매하거나 오유가 있어서는 안된다. 콤퓨터는 정확치 못한 서술에 대하여서는 그 의미를 알수 없어 동작하지 못한다.

다음의 실례는 프로그람작성언어 C++로 10 × 5 의 결과를 인쇄하는 명령문이다.

```
std::cout << (10 * 5);
```

그러나 사실 이렇게 서술하면 프로그람작성자가 아닌 사람은 10 × 5 의 결과를 인쇄하라는 의미를 인차 알수 없을것이다.

1.3.2 콤파일러의 역할

문제풀이를 서술하는 고수준언어를 콤퓨터체계에서 실현시키려면 먼저 그에 해당한 기계어로 변환하여야 한다. 이 변환공정은 콤파일러에 의해 수행된다. 콤파일러(compiler)는 고수준언어명령문들을 콤퓨터가 리해할수 있는 형식으로 변환하는 프로그람이다. 콤파일러는 그 변환과정에 프로그람작성자에게 문제를 고수준언어로 표현하면서 범한 문법적오유나 의미론적오유에 대하여 통보한다. 이 변환과정은 프랑스어방송원이 영어문서의 내용을 랑독할수 있도록 그 문서를 프랑스어로 바꾸어주는 번역원의 작업과 류사하다.

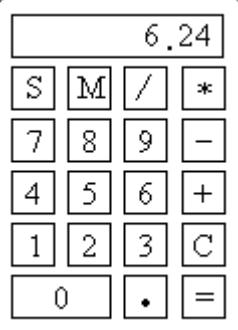
일단 고수준언어로 작성된 프로그람이 콤파일러에 의해 변환된 다음에는 콤퓨터상에서 실행시킬수 있다. 프로그람을 처음 작성하는 사람들은 대부분 자기가 작성한 프로그람의 실행결과가 기대했던대로 되지 않아 놀라군 한다. 콤퓨터는 프로그람명령을 정확히 집행한다. 문제는 프로그람을 처음 작성하는 사람들이 풀이프로그람을 정확히 서술하지 못한데 있다.

1.4 간단한 문제

어느 한 농장상점에서 사과를 판다. 그런데 그 농장상점은 구매상품의 무게만 재고 값은 계산할수 없는 저울을 사용한다. 사과를 산 손님은 구럭에 사과를 가득 채우고 그것을 판매원에게 가져 간다. 판매원은 사과의 무게를 결정하기 위해 저울에 달쿤 다음 kg당 값에 그 무게를 곱한다.

만일 상점판매원이 산수암산을 잘한다면 자기 머리로 계산할수 있지만 그렇지 않다면 사과값을 계산하기 위해 다른 수단을 사용할수 있다.

1.5 전자수판을 리용한 문제풀이

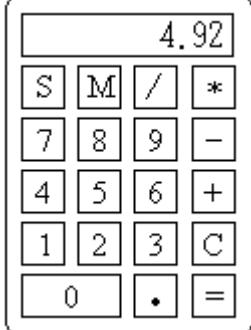
실례로 휴대용전자수판을 사용하여 1kg당 1.20파운드인 사과 5.2kg의 값을 계산하는 아주 간단한 문제를 풀어 보자. 이것은 다음의 4단계로 수행된다.

전자수판	단계	수행단계들
	1	1kg에 해당한 사과값을 입력한다. C 1.20
	2	수행되여야 할 연산자를 입력한다. *
	3	손님이 산 kg수를 입력한다. 5.2
	4	계산단추를 누른다. =

주의: 전자수판에서 건들의 기능은 다음과 같다.

C 현시부의 내용을 지우거나 전자수판이 꺼 있는 경우에는 켠다.
S 기억기에 현시부내용을 보관한다.
M 기억기의 내용을 복귀한다.
+ - * / 산수연산자들
= 계산

우의 단계들을 수행하면 1.20×5.2에 대한 값이 표시된다. 문제를 푸는데서 그 문제는 여러개의 아주 간단한 단계들로 갈라 진다. 이 단계들이 바로 전자수판이 리해하는 《언어》이다. 이 단순한 지령들에 따라 전자수판은 1kg당 1.20파운드인 사과 5.2kg의 값을 구하는 문제를 《푼다》.

1.5.1 보다 일반적인 풀이법

앞에서 설명한 계산을 전자수판의 기억기에 사과값을 기억시키는 방법으로 좀 더 일반적으로 할수 있다. 일정한 량에 대한 사과값은 기억되여 있는 사과값을 불러낸 다음 여기에 요구되는 량을 곱하여 계산할수 있다. 실례로 4.1kg의 사과값을 계산하기 위하여 다음과 같은 수행단계를 걸친다.

전자수판	단계	수행단계들
	1	1kg에 해당한 사과값을 입력한다. C 1.20
	2	전자수판의 기억기에 이 값을 기억시킨다. S
	3	기억기로부터 그 값을 불러 낸다. M
	4	수행되여야 할 연산자를 누른다. *
	5	손님이 산 kg수를 입력한다. 4.1
	6	계산단추를 누른다. =

매 손님들의 사과주문값을 계산하려면 3-6단계만 반복하면 된다. 따라서 어떤 량에 대한 사과값을 구하는 문제의 일반적인 풀이가 정의되고 실현되였다.

1.6 C++언어를 리용한 문제풀이

프로그람작성언어 C++를 리용하여 사과값을 계산하는 문제풀이는 앞에서 본 전자수판에 의한 처리방법과 비슷하다. 이때 개별적인 단계들은 다음과 같다.

단계	설 명
1	기억구역 price_per_kilo에 kg당 사과값을 설정한다.
2	기억구역 kilos_of_apples에 요구된 사과무게(kg)를 설정한다.
3	기억구역 price_per_kilo의 내용에 기억구역 kilos_of_apples의 내용을 곱한 결과를 기억구역 cost에 설정한다.
4	기억구역 cost의 내용을 인쇄한다.

주의: 1.2×5.2만을 계산하면 프로그람이 더 짧게 작성될수 있지만 우의 풀이법은 임의의 무게에 대한 사과값을 계산하는데로 쉽게 확장될수 있다.

대부분의 프로그람작성언어와 마찬가지로 C++에서도 값을 기억시키는데 필요한 기억구역이 미리 선언되여야 한다. 이 기억구역을 선언하는데는 많은 리유들이 있는데 그중 일부 리유는 다음과 같다.

- 이 기억구역에 기억되여야 할 항목들의 형을 지정하기 위해서이다. 기억해야 할 항목형들을 지정함으로써 콤파일러는 프로그람작성자가 우연히 놓쳤거나 타당치 않은 항목을 기억위치에 기억시키는것을 검사할뿐아니라 항목들에 대한 정확한 기억기량을 할당할수 있다.
- 프로그람작성자가 우연히 기억구역 cost를 c0st로 타자한 경우에 c0st에 값을 기억시키지 않도록 하기 위해서이다. 그것은 영문자 《o》를 수자 령 《0》으로 잘못 쳤기때문이다.

우에서 서술한 단계들을 개별적인 C++명령문들로 콤파일한 다음 콤퓨터에서 실행시키면 화면에 1 kg당 1.20 파운드인 사과 5.2kg 의 값이 표시된다.

단계	행	C++명령문들
	1	**double** price_per_kilo;
	2	**double** kilos_of_apples;
	3	**double** cost;
1	4	price_per_kilo = 1.20;
2	5	kilos_of_apples = 5.2;
3	6	cost = price_per_kilo * kilos_of_apples;
4	7	std::cout << cost << “ \n ” ;

주의: 굵은체로 쓴 단어들은 C++언어의 예약어이며 기억구역의 이름으로 사용할수 없다.
기억구역의 이름을 쉽게 알아 볼수 있도록 이름에 《_》를 포함시킨다.
기억구역의 이름에는 공백을 넣을수 없다.
모든 C++명령문은 《;》으로 끝난다.
곱하기는 《*》로 쓴다.
C++에서 행바꾸기기호는 "\n"으로 표시한다.

C++프로그람코드의 매행들은 다음의 동작들을 수행한다.

행	설 명
1	사과의 1kg당 값을 보관할 price_per_kilo라는 기억구역을 할당한다. 이 기억구역은 double형이며 소수도 가질수 있다.
2-3	기억구역 kilos_of_apples와 cost를 할당한다.
4	기억구역 price_per_kilo의 내용을 1.20으로 설정한다. =는 《값이 할당되다.》라는 뜻이다.
5	기억구역 kilos_of_apples에 5.2를 넣는다.
6	기억구역 price_per_kilo의 내용을 기억구역 kilos_of_apples의 내용으로 곱한 다음 그 결과를 기억구역 cost에 설정한다.
7	기억구역 cost의 내용을 콤퓨터화면에 표시한다. 이 명령문의 구성요소들의 의미는 다음과 같다. 출력흐름 → std::cout, 인쇄될 기억구역 → cost, 행바꾸기 → "\n" std::cout << cost << "\n"; 이것은 《출력흐름 std::cout 에 기억구역 cost의 내용을 삽입하고 다음행으로 이동하시오.》로 해석할수 있다.

이 풀이법은 기억된 값들을 보존하기 위하여 매 기억구역들이 리용되는것을 제외하고는 휴대용전자수판을 사용한 풀이법과 거의 비슷하며 그 계산이 사람들이 더 쉽게 해석할수 있는 형식으로 서술된다.

우의 C++프로그람의 동작과정을 아래에서 보여 준다. 동작과정에서는 매 C++명령문을 실행하였을 때 기억구역의 내용이 어떻게 되는가를 보여 준다. C++에서는 기억구역이 함수안에서 선언될 때 그의 초기내용은 정의되지 않는다.

C++ 명령문들	price △	kilos △	cost
double price_per_kilo; **double** kilos_of_apples; **double** cost;	U	U	U
price_per_kilo = 1.20;	1.20	U	U
kilos_of_apples = 5.2;	1.20	5.2	U
cost = price_per_kilo * kilos_of_apples;	1.20	5.2	6.24
std::cout << cost << "\n";	1.20	5.2	6.24

주의: U는 기억구역의 내용이 정의되지 않았다는것을 표시한다.
△은 표제란에 있는 칸이 작아서 변수 price_per_kilo 를 price 로, kilos_of_apples 를 kilos 로 표시하기 위하여 쓰이였다.

1.6.1 프로그람의 실행

우에 있는 코드는 완성된 C++프로그람이 아니며 프로그람의 기본부분이다. 이 코드에 프로그람을 보충, 완성한 다음 실행하면 다음과 같은 결과가 나온다.

```
6.24
```

프로그람의 내용을 아는 사람이라면 즉시에 이것이 1kg당 1.20파운드인 사과 5.2kg의 값을 나타낸다는것을 알수 있을것이다. 그러나 그 내용을 모르는 사람들은 이것이 무엇을 의미하는지 알수 없을것이다.

1.7 설명문이 노는 역할

C++프로그람에 설명문(comment)들을 달아 주면 프로그람을 해석하기가 쉬워진다. 설명문은 //으로 시작되며 그 행의 마지막까지를 설명문으로 인식한다. 프로그람에 씌여 진 설명문들은 해석에 도움을 줄뿐이며 프로그람이 실행될 때 콤퓨터는 그 설명문들을 완전히 무시한다. 실례로 우의 코드에는 다음과 같은 설명문을 달수 있다.

```
double price_per_kilo;                       // kg당 사과값
double kilos_of_apples;                      // 요구되는 사과량
double cost;                                 // 사과값

price_per_kilo   = 1.20;                     // kg당 값 1.20 을 설정
kilos_of_apples = 5.2;                       // 요구되는 kg

cost = price_per_kilo * kilos_of_apples;     // 값계산
std::cout << cost << “ \n ”;                 // 값인쇄
```

주의: 이것은 설명문의 한 실례이다. 경험 있는 프로그람작성자들은 아마 프로그람을 쉽게 리해할수 있으므로 우에 있는 설명문중 대부분을 뺄것이다.

프로그람코드에 대한 독자들의 리해를 도모하지 못하는 설명문들은 피해야 한다. 일부 경우에는 기억구역이름을 그 의미가 충분히 알리도록 달아 준다. 일반적으로 프로그람내용을 인차 리해할수 없을 때 설명문을 달아 준다.

1.8 요 약

지금까지 보여 준 C++명령문들은 다음과 같다.

C++명령문	설 명
double cost;	cost 라는 기억구역을 선언한다.
cost = 1.2 * 5.2;	기억구역 cost 에 1.2*5.2 의 계산결과를 넣는다.
std::cout << "Hi !";	통보문 Hi !를 인쇄한다.
std::cout << cost << "\n";	기억구역 cost 의 내용을 인쇄하고 행바꾸기한다.

이 형식의 명령문들을 가지고도 여러가지 쓸모 있는 프로그람들을 작성할수 있다.

1.9 보다 구체화된 프로그람

프로그람에 출력명령들을 더 추가하면 사용자들은 그 프로그람의 결과를 쉽게 알수 있다. 실례로 1.6 에서 보여 준 프로그람은 아래의 프로그람으로 바꿀수 있다. 이 프로그람의 기본부분은 사용자가 결과들이 무엇을 의미하는지 알수 있도록 하는데 중점을 두고 있다.

행	C++명령문들
1	**double** price_per_kilo; // kg당 사과값
2	**double** kilos_of_apples; // 요구되는 사과량
3	**double** cost; // 사과값
4	price_per_kilo = 1.20;
5	kilos_of_apples = 5.2;
6	cost = price_per_kilo * kilos_of_apples;
7	std::cout << "Cost of apples per kilo £ ";
8	std::cout << price_per_kilo;
9	std::cout << "\n";
10	std::cout << "Kilos of apples required K ";
11	std::cout << kilos_of_apples;
12	std::cout << "\n";
13	std::cout << "Cost of apples £ ";
14	std::cout << cost;
15	std::cout << "\n";

행	설 명
1-6	1kg당 1.20파운드인 사과 5.2kg에 대한 값계산.
7	통보문 Cost of apples per kilo ￡를 화면에 표시. 통보문앞뒤에 있는 인용부호는 이것이 기억구역의 내용이 아니라 인쇄되여야 할 통보문이라는것을 표시하는데 쓰인다.
8	우의 통보문다음에 기억구역 cost의 내용을 화면에 표시한다.
9	화면에서 행바꾸기를 진행한다.
10-12	7 - 9행과 같은데 이번에는 통보문이 kilos of apples required K 이고 기억구역 kilos_of_apples의 값을 화면에 표시한다.
13-15	7 - 9행과 같은데 이번에는 통보문이 Cost of apples ￡이고 기억구역 cost의 값을 화면에 표시한다.

1.9.1 한 명령문으로 여러 항목들을 인쇄하기

```
std::cout << “Cost of apples          £ ” ;
std::cout << cost;
std::cout <<  “ \n ” ;
```

우의 명령문들은 다음과 같이 한개의 명령문으로 합쳐 질수 있다.

```
std::cout << “Cost of apples          £ ” << cost << “\n” ;
```

1.9.2 새 프로그람을 실행

우의 프로그람을 보충, 완성한 다음 실행하면 다음과 같은 결과가 나온다.

```
Cost of apples per kilo    £ 1.2
Kilos of apples required   K 5.2
Cost of apples             £ 6.24
```

이것을 보면 프로그람이 무엇을 계산해 냈는지 인차 알수 있다.

1.10 기억구역의 종류

지금까지 기억구역형들은 double형으로 되여 있었다. double형의 기억구역에는 소수부를 가진 임의의 수를 넣을수 있다. 그러나 이러한 형으로 선언하면 기억구역에는 소수부를 가지는 수만이 기억된다. 사실 정확히 옹근값으로 표현되는 수를 써야 하는 경우도 제기된다. 실례로 기억구역 people에 방에 들어 있는 사람수를 넣을수 있다. 이때 기억구역은 int형으로 선언되여야 한다. 실례로 다음의 코드는 방에 있는 사람수를 기억시키기 위해 int형기억구역을 사용한다.

```
int  room;                                  // 기억구역
room = 7;                                   // 수 7 을 넣는다.
```

기억구역형의 선택은 물론 그 기억구역에 기억시키려는 값에 의존한다. 일반적으로 정확한 옹근수가 요구된다면 int 형의 기억구역을 사용하고 그 값이 소수부를 가져야 한다면 double 형의 기억구역을 사용한다.

기억구역	기억구역에 값주기
int people;	people = 2;
double weight;	weight = 7.52;

1.11 반 복

지금까지 실례들에서 모든 C++프로그람들은 행들이 일직선으로 되여 있었다. 행이 일직선으로 된 코드에서는 프로그람이 우에서부터 아래로 가면서 한개 명령문씩 수행한다. 그 프로그람에서 조종흐름에 영향을 미치는 명령문은 없다. 이 방법은 kg당 1.20파운드인 사과 5.2kg의 값계산이라는 지정된 경우에 대하여 쓸수 있다.

이 방법을 사용하여 각이한 무게에 대한 사과값을 렬거하는 프로그람을 작성하려면 실제로 같은 코드를 여러번 서술해야 할것이다. 이러한 형태로 프로그람을 작성하면 다음과 같다.

```
double price_per_kilo;                            // kg당 사과값
double kilos_of_apples;                           // 요구되는 사과량
double cost;                                      // 사과값
price_per_kilo = 1.20;
std::cout << “ Cost of apples per kilo       :”;
std::cout <<   price_per_kilo     << “\n”;
std::cout << “Kilo’s Cost ” << “\n”;
```

```
kilos_of_apples            = 0.1;
```

```
cost = price_per_kilo * kilos_of_apples;
std::cout << kilos_of_apples << “        ”<< cost <<  “\n”;
```

```
kilos_of_apples             = 0.2;
```

```
cost = price_per_kilo * kilos_of_apples;
std::cout << kilos_of_apples << “        ”<< cost << “\n”;
```

이런 식으로 하게 되면 100개의 각이한 무게들의 값을 계산하는 경우 품이 많이 들고 코드를 많이 작성해야 한다. 코드를 입력해 넣는 품은 제외하고라도 프로그람 편집만 하자고 해도 복사 및 붙이기조작을 비롯해서 상당한 품을 들여야 한다. 결과적으로 프로그람은 커지며 상당한 자원을 소비할것이다.

1.12 while 명령문

C++에서 while명령문은 조건이 참인 동안 프로그람명령문들을 반복한다. while명령문은 그림 1-1에서 보여 주는바와 같이 기차길처럼 되여 있다. 조건이 참인 동안 조종흐름은 《참》의 길을 따라 간다. 순환주기에서 매번 조건이 재평가된다. 그 다음 조건이 거짓이라는것을 발견할 때는《거짓》의 길을 따라 간다.

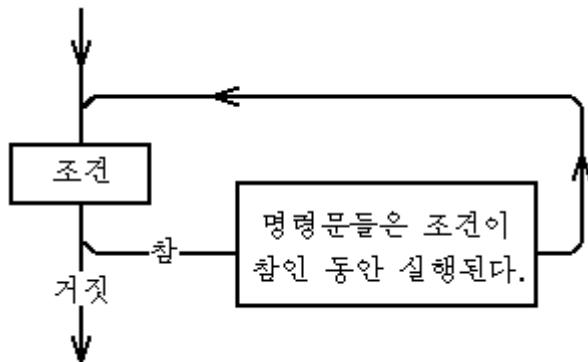



그림 1-1. 기차길과 같은 while명령문

while순환에서는 항상 조건이 먼저 검사된다. 이 요구때문에 처음에 조건이 거짓으로 평가된다면 while순환과 결합된 코드는 전혀 실행되지 않는다.

1.12.1 조건

C++언어에서 조건은 매우 간결한 형식으로 표현된다. 실례로 조건식 《기억구역 count의 내용은 5보다 작거나 같다.》를 쓰면 다음과 같다.

```
count <= 5
```

주의: count 이름으로 된 기억구역은 다음과 같이 선언되여야 한다.

```
int count;
```

조건에서 사용되는 기호들은 다음과 같다.

기호	의미	기호	의미
<	작기	<=	작거나 같기
==	같기	!=	같지 않기
>	크기	>=	크거나 같기

만일 기억구역에 아래의 값이 들어 있는 경우

기억구역	값주기
int temperature;	temperature = 15;
double weight;	weight = 50.0;

다음의 표는 C++로 씌여 진 여러 조건식들의 참 혹은 거짓값을 보여 준다.

설 명	조 건	결과
온도는 20 ℃보다 높다.	temperature < 20	true
온도는 20 ℃이다.	temperature == 20	false
무게는 30 kg과 같거나 더 무겁다.	weight >= 30.0	true
20 ℃는 정해 진 온도보다 낮은 값이다.	20 < temperature	false

주의: double형값을 가진 기억구역에 정확한 수가 들어 있을 때 ==(같기)나 !=(같지 않기)로 값을 비교하는것은 좋은 방법이 못된다.

1.12.2 C++에서 while 명령문

while 명령문을 리용하여 본문통보문 Hello 를 5 번 써내는 코드를 작성하면 다음과 같다.

```
int count;
count = 1;                          // count 를 1 로 설정
while ( count <= 5 )                // count 가 5 보다 작거나 같아 지는 동안
{
   std::cout << "Hello" << "\n";    // Hello 인쇄
   count = count + 1;               // count 에 1 을 더하기
}
```

주의: 명령문 count = count + 1;은 count의 내용에 1을 더하고 기억구역 count에 그 결과값을 다시 넣는다.

이 코드에서 괄호 { }사이에 있는 명령문들은 count내용이 5보다 작아 지는 동안 반복적으로 실행된다. while명령문의 조종흐름을 그림 1-2에서 보여 준다.

count <= 5

std::count << "Hello" << "\n";
count = count + 1;

참

거짓

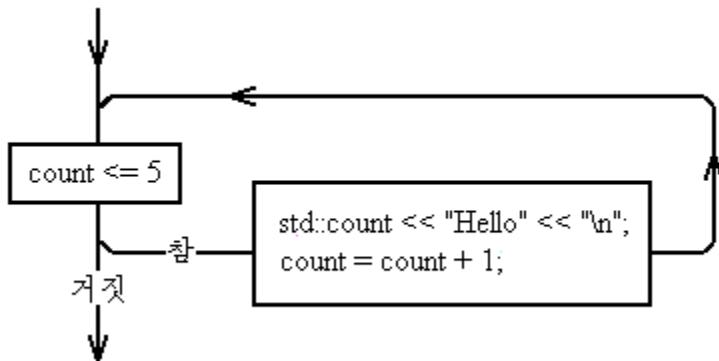



그림 1-2. C++에서 while명령문의 조종흐름

1.12.3 while명령문의 사용법

콤퓨터프로그람을 리용하는 실제적인 우점은 작성된 코드를 수많이 반복해야 하는 경우에 나타나며 이때 프로그람을 리용하면 상당한 시간과 품을 덜수 있다. 실례로 각이한 무게에 대한 사과값을 렬거하는 프로그람을 작성하면 어떤 지정된 무게에 해당하는 사과값을 계산하는 C++코드행들을 반복하는것으로 구조화된다. 이 계산을 매번 반복할 때마다 사과무게가 들어 있는 기억구역의 내용은 변화된다. 이것을 계산하는 C++코드는 다음과 같다.

```
double price_per_kilo;                          // kg당 사과값
double kilos_of_apples;                         // 요구되는 사과량
double cost;                                    // 사과값

price_per_kilo = 1.20;

std::cout << “Cost of apples per kilo  :”;
std::cout << price_per_kilo      << “\n”;

std::cout << “Kilo’s Cost” << “\n”;
kilos_of_apples = 0.1;

while ( kilos_of_apples <= 10.0 )               // 인쇄하려는 행수만큼 진행
{
   cost = price_per_kilo * kilos_of_apples;     // 값계산

   std::cout << kilos_of_apples;                // 결과인쇄
   std::cout << “          ”;
```

```
    std::cout << cost << "\n";

    kilos_of_apples = kilos_of_apples + 0.1;        // 다음값
}
```

우의 프로그람을 보충, 완성한 다음 실행하면 다음과 같은 결과가 나온다.

```
Cost of apples per kilo  : 1.2
Kilo's Cost
0.1     0.12
0.2     0.24
0.3     0.36
0.4     0.48
0.5     0.6
0.6     0.72
0.7     0.84
0.8     0.96
…
9.9     11.88
10.0    12.0
```

주의: 수의 출력형식들을 조종하기 위한 명령문들이 보충적으로 필요하다. 수의 출력형식을 조종하는 출력조작자들에 대해서는 부록 1 을 보시오.

1.13 선 택

if구조는 조건결과에 따라 명령문들을 실행하는데 쓰인다. 이 명령문은 그림 1-3에서 보여 준 기차길과 비슷한데 어느 통로를 취하겠는가 하는것은 조건의 결과에 따른다. 그러나 while명령문과 달리 조건부분을 반복실행하지는 않는다.

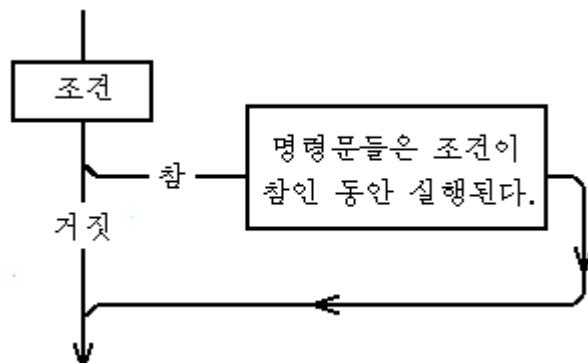



그림 1-3. 기차길처럼 표현된 if 명령문

실례로 다음의 C++프로그람부분은 기억구역 temperature의 내용이 30보다 클 때 Hot!를 인쇄한다.

```
int  temperature;
temperature = 30;

if ( temperature > 30 )                    // 만일  temperature 의 내용이 30 보다  크면
{
    std::cout << “Hot! ” << “\n”;                         // 덥다
}
```

이 코드에서 괄호 { }사이에 있는 명령문들은 조건 temperature>30이 참일 때에만 실행된다. 우의 코드에 대한 조종흐름을 그림 1-4에서 보여 준다.

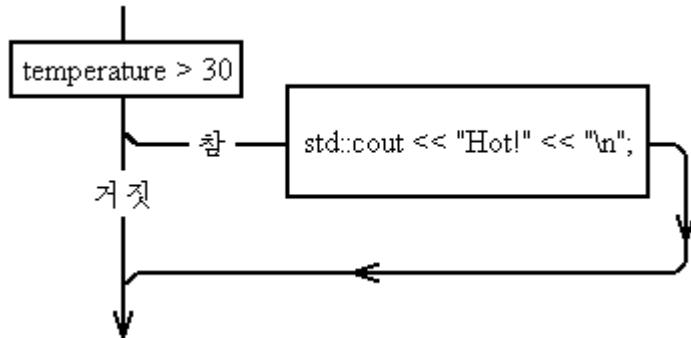



그림 1-4. 기차길처럼 표현된 if 명령문

1.13.1 if 명령문의 사용법

각이한 무게에 대한 사과값을 렬거하는 프로그람을 인쇄할 때 결과를 5개씩 구분하는 공백행을 삽입하면 보기 좋을것이다. 이것은 인쇄행수를 세는 계수기 lines_output를 써서 마지막 5번째 행에서는 공백행을 삽입하게 하여 실현할수 있다. 그 공백행을 인쇄한 다음 계수기 lines_output를 0으로 재설정한다. 프로그람을 수정하면 다음과 같다.

```
double price_per_kilo;                          // kg당 사과값
double kilos_of_apples;                         // 요구되는 사과량
double cost;                                    // 사과값

price_per_kilo   = 1.20;
kilos_of_apples = 0.1;

std::cout << “Cost of apples per kilo : ”;
std::cout << price_per_kilo << “\n”;
```

```
std::cout << “Kilo’s Cost”;
kilos_of_apples = 0.0;
int lines_output = 0;

while ( kilos_of_apples <= 10.0 )                    // 인쇄하려는 행수만큼 진행
{
    cost = price_per_kilo * kilos_of_apples;         // 값계산

    std::cout << kilos_of_apples;                    // 결과인쇄
    std::cout << “      ”;
    std::cout << cost << “\n”;

    kilos_of_apples = kilos_of_apples + 0.1;         // 다음값
    lines_output = lines_output + 1;                 // 1을 더하기

    if ( lines_output >= 5 )                         // 인쇄묶음
    {
        std::cout << “\n”;                           // 빈 행인쇄
        lines_output = 0;                            // 계수기의 재설정
    }
}
```

우의 프로그람을 보충, 완성한 다음 실행하면 다음과 같은 결과가 나온다.

```
Cost of apples per kilo  : 1.2
Kilo’s Cost

0.0    0.0
0.1    0.12
0.2    0.24
0.3    0.36
0.4    0.48

0.5    0.6
0.6    0.72
0.7    0.84
0.8    0.96
0.9    1.08

1.0    1.2
1.1    1.32
1.2    1.44
1.3    1.56
1.4    1.68

…
```

1.14 자체평가

- 콤퓨터프로그람작성언어란 무엇인가?
- 다음의 C++코드의 실행결과는 무엇인가?

```
int i;
i = 10;
while ( i > 0 )
{
    std::cout << i;
    i = i - 1;
}
std::cout << “ \n ” ;
```

```
int temperature;
temperature = 10;
if ( temperature > 20 )
{
    std::cout << “It’s Hot! ” ;
}
if ( temperature <= 20 )
{
    std::cout << “It’s not so Hot! ” ;
}
std::cout << “ \n ” ;
```

- 다음의 C++코드에서 틀린것은 무엇인가?

```
int value = 3;
if ( value = 2 )
{
    std::cout << “Value is equal to 2 ” ;
}
```

- 다음의 조건들에 대해 C++조건식으로 쓰시오. 사용될 기억구역들을 어떻게 선언해야 하는가?
  - ✧ 온도가 15 ℃보다 낮다.
  - ✧ 대학까지의 거리는 15km 보다 짧다.
  - ✧ 대학까지의 거리는 축구장까지의 거리와 같거나 멀다.
  - ✧ 자전거값이 록화기값과 같거나 눅다.

1.15 종이우에서의 련습

C++명령문들로 다음의 문제들을 푸는 과정을 학습장에 쓰시오. 이 풀이는 콤퓨터에서 실행하지 않아도 된다.

- 이름
  자기의 이름과 주소를 쓰시오.
- 무게
  27개 지함의 전체 무게를 계산하시오. 매 지함의 무게는 2.4㎏이다.
- 이름
  while순환을 사용하여 통보문 "Happy Birthday"를 3번 출력하시오.
- 구구표
  7계단 구구표를 출력하시오. 출력형식은 다음과 같다.

```
7*1=7
7*2=14
. . .
```

참고: 먼저 int형변수 row에 값 3을 넣고 3번째 행을 인쇄하기 위한 C++코드를 쓰시오.

```
7 * 3 = 21
```

다음으로 row의 내용을 1부터 9까지 변화시키는 순환속에 이 명령문들을 넣으시오.

- 무게표
  매 지함의 무게가 2.4㎏일 때 1부터 20개 지함의 무게를 렬거하는 표를 인쇄하시오.

- 구구표
  1부터 5까지의 모든 값들에 대한 구구표를 인쇄하시오. 다음과 같은 표를 만드시오.

```
       |  1    2    3    4    5
-------+---------------------------
       |
  1    |  1    2    3    4    5
  2    |  2    4    6    8    10
  3    |  3    6    9    12   15
  4    |  4    8    12   16   20
  5    |  5    10   15   20   25
```

참고: 먼저 int형변수에 값 2를 넣고 두번째 행을 인쇄하는 C++코드를 쓰시오.

```
2  /  2   4   6   8   10
```

row의 내용을 1부터 5까지 변화시키는 순환속에 이 명령문들을 넣으시오.
그 다음 프로그람에 제일 웃행을 인쇄하는 명령문들을 추가해 넣으시오.

```
   /  1   2    3   4    5
------------------------
```

2 쏘프트웨어설계

이 장에서는 쏘프트웨어생산의 전반과정을 보여 준다. 특히 크고작은 쏘프트웨어체계개발에서 제기되는 문제점들에 대하여 서술한다. 그리고 콤퓨터상에서 실현되는 문제에 대한 해답을 문서화하고 표시하는 도구인 UML(Unified Modelling Language, 통합모형화언어)에 의한 표기법을 소개한다.

2.1 쏘프트웨어위기

콤퓨터가 쓰이던 초시기 하드웨어는 값이 매우 비쌌다. 이런 콤퓨터에서 실행되였던 프로그람들은 지금의 일반프로그람들보다 엄청나게 작았다. 또한 콤퓨터들의 주기억장치(random access memory)와 보조기억장치(disk storage)의 용량이 매우 제한되여 있었다.

그후 전변이 일어 났다. 기술의 진보는 이전의 기계보다 훨씬 더 용량이 크고 값이 눅은 콤퓨터들을 만들수 있게 하였다. 쏘프트웨어개발자들은 생각하였다. 《좋다! 더 크고 더 전면적인 프로그람들을 만들수 있다.》 쏘프트웨어설계들은 넓은 범위에서 아주 락관적으로 발전하기 시작하였다.

그러나 얼마 안 있어 설계들이 비용이 많이 들고 주문자들의 요구를 충족시킬수 없게 되자 대규모의 쏘프트웨어개발이 힘들다는것이 분명해 졌다. 그리고 소규모의 쏘프트웨어개발에 사용된 초기의 기술로 대규모의 쏘프트웨어들을 성과적으로 만들수 없었다.

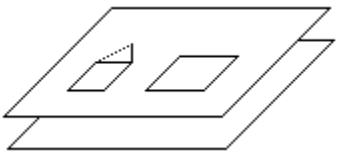
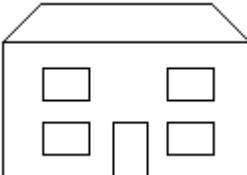
이것은 자전거려행과 비슷하다. 짧은 거리에는 자전거려행이 맞을수 있어도 먼 거리를 짧은 시간동안에 해야 하는데는 자전거려행이 맞지 않는다.

2.2 문제와 모형, 풀이법

문제를 풀기 위해서는 먼저 그 문제를 리해해야 한다. 문제를 충분히 리해해야 그 풀이법을 공식화할수 있다.

문제를 리해하고 푸는 방법에는 여러가지가 있다. 일반적으로 문제를 푸는 방법은 표준표기법이나 프로그람작성자가 창안한 표기법을 사용하여 문제와 그 풀이과정을 서술하는것이다. 표준표기법을 사용하는 우점은 다른 사람들이 그 문제의 서술과 제기된 풀이과정을 검사하거나 수정할수 있다는것이다. 실례로 집을 짓는다고 할 때 먼저 건축가는 건설될 여러 부분들에 대한 설계도들을 그린다. 주문자는 그 설계도들을 정식 승인하거나 그 일부를 수정하여 적당히 승인할수 있다. 건설자들은 집을 건설할 때 그 설계도들을 리용한다.

콤퓨터프로그람작성은 이와 비슷한 처리를 진행한다. 먼저 콤퓨터프로그람이 수행해야 할 과제를 리해하여야 한다. 그다음 모형을 만들고 풀이를 실현해야 한다.

전문가의 설계(모형)	완성된 집

이때 작은 문제풀이에 사용된 모형으로 큰 문제를 풀수 있다고 생각하는것은 잘못된 생각이다. 실례로 자그마한 개울을 넘기 위해서는 통나무다리를 놓고 건느든가 다리를 놓지 않고 그저 뛰여 넘을수도 있다. 그러나 개울을 건느는 이 방법으로 큰 강을 건늘수는 없을것이다. 마찬가지로 100층짜리 고층건물을 건설하기 위하여 건축가는 2층집설계도를 가지고 건설자들에게 나머지층을 더 건설하라고 단순하게 지시하지 못할것이다.

쏘프트웨어에서도 규모(scale)문제가 존재한다. 일반적으로 작은 프로그람을 실현하기 위하여 사용한 방법이 큰 프로그람작성설계에 사용될수 없다. 문제에 대한 충분한 파악이 없이 체계설계를 시작한다면 그 쏘프트웨어는 많은 재난을 낳게 될것이다.

2.2.1 책임

오래전부터 사람들은 책임을 가지고 있다고들 말해 왔다. 어렸을 때는 이 책임이 아주 간단하지만 나이가 들수록 점점 늘어 난다. 책임(responsibility)이란 무엇인가를 돌봐야 하는 짐이거나 신임 혹은 의무이다. 어렸을 때에는 이것이 자기 방을 산뜻하고 깨끗하게 거두는것으로 될수 있다. 나이가 들수록 책임이 가지는 항목들의 범위나 복잡성은 상당히 늘어 날것이다.

학생에 대하여 실례를 든다면 학생은 배우는 학과목을 리해하는데 책임을 가진다. 교원은 명백하고 리해력 있는 방법으로 학생들에게 학과목을 배워 주는 책임을 가진다. 학생과 교원의 책임들을 요약해 보면 다음과 같다.

학생의 책임	교원의 책임
배우는 학과목을 리해한다.	학과목을 배워 준다.
학과목시험을 잘 치르기 위해 최대의 능력을 낸다.	학과목시험을 치우고 점수를 매긴다.
	학과목시험점수를 발표한다.

쏘프트웨어 역시 책임을 가지고 있다. 실례로 본문편집기는 사용자가 작성한 본문을 정확히 문서로 입력하는 책임을 가진다. 만일 본문이 정확치 않거나 무의미하게 입력되였다면 결과문서도 정확하지 않을것이다. 본문편집기는 입력된 본문의 총체적인 타당성에 대하여 책임을 지지 않는다.

일찌기 콤퓨터계에서는 《정확치 못한 자료가 들어 가면 정확치 못한 결과가 나온다(garbage in, garbage out).》라는 말을 써왔다. 프로그람묶음이 자기 책임을 정확하게 실현하였다고 하여도 정확치 못한 자료가 들어 오면 그 결과는 무의미하거나 지어는 손해를 줄수 있다.

2.3 객체

우리가 살고 있는 세계는 수많은 각이한 객체(object)들로 이루어 져 있다. 실례로 사람들은 보통 다음의 객체들중 적어도 일부를 사용한다.

- 전화기
- 콤퓨터
- 차

매 객체는 자기의 책임들을 가지고 있다. 실례로 우의 객체와 련관된 책임들중 그 일부는 다음과 같다.

객 체	책 임
전화기	• 다른 전화지점과 련결한다. • 음성과 전기적신호와의 호상변환을 실현한다.
콤퓨터	• 프로그람을 실행한다. • 인터네트에 tcp/ip접속을 제공한다.
차	• 움직인다. • 더 빨리/더 천천히 간다. • 왼쪽/오른쪽으로 방향을 전환한다. • 정지한다.

여기서 책임은 객체가 수행하는 처리이다. 실례로 차는 앞으로 혹은 뒤로 움직일수 있다. 그러나 차를 모는것은 운전수이다. 객체는 피동이며 지시에 의해서만 움직인다.

2.3.1 객체로서의 차

차는 많은 부분품들 혹은 객체들로 되여 있다. 사용자의 시점에서 볼 때 차를 이루는 주요객체들은 다음과 같다.

- 차체(본체)
- 기관
- 변속치차함
- 크라치
- 전원을 공급하는 축전지

모든 객체들이 들어 있는 용기(container)로서 차체를 생각할수 있고 그것들이 조립되면 움직이는 차가 만들어 진다고 생각할수 있다. 이 각이한 객체들은 차안에 있는 운전수에게는 보이지 않는다. 그러나 운전수는 차체의 부분인 외부대면부들을

사용하여 이 객체들과 호상작용한다. 이 객체들의 배렬을 UML표기법을 리용하여 표시하면 그림 2-1과 같다.

승용차본체
1 1 1 1
1 1 1 1
기관 변속치차함 크라치 축전지

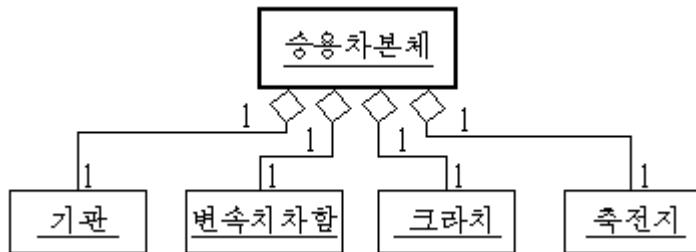



그림 2-1. 차를 이루는 객체들

그림 2-1에서는 아래의 표기법형식이 사용된다.

기관	객체를 의미한다. 여기서는 차의 기관이다.
기관 기야통	집합체를 의미한다. 기관은 4개의 피스톤들로 이루어 졌다. **주의**: 집합체(aggregation)를 나타낸다. 즉 용기 A에 부분품 B가 들어 있다. A B 조립체(composition)를 나타낸다. 즉 용기 A는 부분품 B를 창조하거나 파괴한다. A B

이 표기법을 사용하여 객체들에 대하여 《부분》관계를 표시할수 있다. 기관과 변속치차함, 크라치와 축전지는 차의《부분》이다.

2.4 클라스

객체지향전문용어에서 클라스는 같은 책임들과 같은 내부구조를 공유하는 모든 객체들을 표현하는데 사용된다. 클라스(class)란 본질에 있어서 같은 종류의 객체들에 대한 집합적인 이름이다. 실례로 다음의 객체들은 모두 차(Car)클라스에 속한다.

A의 빨간 승용차	B의 은색 승용차	C의 푸른 승용차

이 객체들은 세부적으로 서로 다를지라도 같은 내부구조와 책임을 가진다. 매 객체는 Car클라스의 구체례이다. 클라스와 객체의 표기법은 좀 차이난다. 클라스와 객체에 대한 UML표기법은 다음과 같다.

클라스	객체(클라스의 구체례)
승용차	A의 승용차

주의: 객체이름은 밑선으로 그어 준다.

클라스와 객체를 구별하는것은 중요하다. 간단히 설명한다면 객체들은 보통 물리적인 표현을 가지는 반면에 클라스들은 추상적인 개념이라는것이다.

2.5 메쏘드와 통보문

메쏘드(method)는 클라스의 책임을 실현한다. 실례로 클라스 Car의 책임들중 일부는 다음과 같다.

- 기관의 시동/정지
- 더 빨리/더 천천히 가기
- 왼쪽/오른쪽으로 방향전환
- 정지

클라스 Car의 구체례는 객체이다. 객체에 통보문(message)을 보내며 객체안에 있는 은페된 메쏘드(Car클라스의 책임)가 통보문을 처리하기 위해 호출된다. 실례로 가속장치를 누르는것에 의해 승용차운전수는 《더 빨리 가시오.》라는 통보문을 보낸다. 이 통보문의 실현은 기관에 휘발유를 더 보내는 기관조종체계에 의해 진행된다. 그러나 일반적으로 이 조작세부는 승용차운전수와는 상관 없다.

2.6 클라스객체

지금까지 객체들에 대한 용기로서 차체를 보았는데 여러개의 콤퓨터장치들이나 객체들을 위한 용기로서 무릎형(laptop)콤퓨터를 들수 있다. 무릎형콤퓨터는 다음과 같이 구성되여 있다.

- 건반, 접촉판(touch pad), 현시장치와 같은 외부대면부를 가지고 있는 콤퓨터체(외피)
- 내부디스크구동장치
- 망파일체계
- CPU
- 음성 및 도형처리소편

여기에서 망파일체계는 많은 무릎형콤퓨터들에서 공유되며 무릎형콤퓨터들은 개별적으로 망파일체계에 접근할수 있다. 객체지향전문용어로 망파일체계는 모든 말단형콤퓨터(notebook)들에서 공유되는 클라스객체(class object)이다.

공유되는 객체에 대한 개념에서 중요한것은 클라스의 모든 구체례들이 같은 정보에 접근할수 있다는것이다. 따라서 무릎형콤퓨터의 한개 구체례가 망파일체계상에서 파일을 하나 만든다면 다른 말단들은 이 파일의 내용에 접근할수 있을것이다.

UML 표기법을 사용하여 무릎형콤퓨터의 객체들을 배렬하면 그림 2-2와 같다.

무릎형콤퓨터
1 1 1 1
1 * 1 1
내부디스크 망디스크 CPU 도형처리

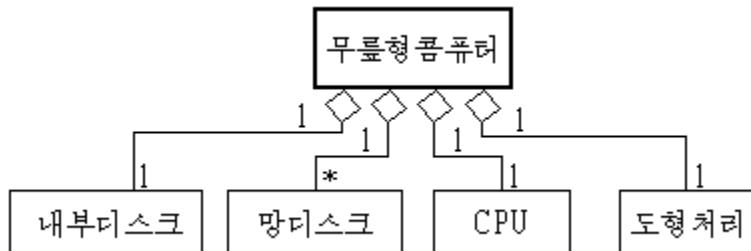



그림 2-2. 사용자의 시점에서 본 무릎형콤퓨터의 객체들

클라스객체의 또 다른 흥미 있는 성질은 용기객체가 요구되지 않는 다른 객체에서 리용할수 있다는것이다. 실례로 망파일체계는 무릎형콤퓨터들이 아닌 다른 장치들에서 사용될수 있다.

2.7 계 승

사무실들에는 일반적으로 다음의 객체가 있다.

- 전화기
- 전화기가 달린 팍스기
- 콤퓨터

이 객체들은 자기고유의 개별적인 책임들을 가진다. 실례로 사무실객체들의 책임중 그 일부는 다음과 같다.

객 체	책 임
전화기	• 다른 전화지점과의 련결 • 전기적신호와 음성신호의 호상변환
전화기가 달린 팍스기	• 다른 전화지점과의 련결 • 전기적신호와 음성신호의 호상변환 • 전기적신호와 영상신호의 호상변환
콤퓨터	• 프로그람들을 실행 • 망에 tcp/ip접속제공

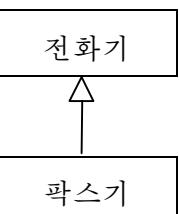
이 책임들을 보면 전화기와 팍스기는 여러개의 책임들을 공유한다. 팍스기는 전화기가 가지고 있는 2가지 책임을 가지고 있다. 팍스기는 영상을 보내거나 받을수 있는 전화기라고 말할수 있다. 팍스기의 또 다른 측면은 바로 그것을 전화기만으로서도 사용할수 있다는것이다. 모든 전화기들과 팍스기들을 표시하는 클라스들사이의 이 관계를 UML표기법을 사용하여 도식으로 보면 그림 2-3과 같다.

계승도	책 임
전화기	다른 전화지점과 련결한다. 음성신호와 전기적신호와의 호상변환을 한다.
팍스기	전화기의 모든 책임들을 포함하며 영상신호와 전기적 신호와의 호상변환도 수행한다.

그림 2-3. 전화기와 팍스기사이의 관계

주의: 상위클라스(전화기)에서 보조클라스(팍스기)가 계승되여 나온다. 계승(inheritance)은 상위클라스의 모든 책임들을 가지게 한다. 다시 말하여 몇개만 선택하여 가질수 없다. 지어는 사용하지 않는 책임일지라도 계승된다.

계승관계는 현재 있는 객체를 전문화(specializing)하여 새로운 객체를 창조할수 있게 하므로 객체지향프로그람작성에서 중요한 개념이다. 새로운 객체의 창조에서 또한 책임들이 구조화되여야 한다. 새로운 쏘프트웨어객체의 개발시간은 이미 있는 책임들을 계승하여 창조함으로써 감소된다. 이러한 공정은 새로운 쏘프트웨어객체들을 창조하는데 요구되는 시간과 품을 훨씬 줄이게 한다.

2.8 다형성

각이한 객체들의 집합에서 모든 객체들이 해당한 통보문을 받을수 있다면 이 통보문은 그 집합에 있는 임의의 객체에 보낼수 있다. 이 통보문을 객체가 받았을 때 실행되는 메쏘드는 통보문을 받은 객체의 형에 의존한다.

실례로 사람들에게 체육활동에 어떻게 참가하는가고 묻는다면 그들은 아마 서로 다르게 대답할것이다. 사실상 통보문 《어떤 체육활동에 참가하는가.》는 그 대답이 질문을 받은 각이한 사람들에 따라 다르므로 다형(polymorphism)적이다. 실례로 정구애호가는 골프애호가와는 다르게 말할것이다.

2.9 자체평가

- 작은 문제에 대한 풀이법을 왜 훨씬 더 크고 복잡한 문제를 푸는데로 확장할 수 없는지 설명하시오.
- 《책임》이란 무엇인가?
- 다음의 객체에서 책임은 무엇인가?
  - ✧ 록화촬영기
  - ✧ 자명종시계
  - ✧ 교통신호등
  - ✧ 체호브의 작품 《세 자매》에서 올가의 역을 수행한 녀배우
- 객체와 통보문, 메쏘드사이에는 어떤 관계가 있는가?
- 다음의 객체들은 어느 클라스들에 속하는가?

아빠트	고양이	크레용	원주필	개
얼럭쥐	얼음집	집	지우개	도서관
큰 저택	사무용품일식	연필	토끼	양

어느 클라스가 다른 어느 클라스의 보조클라스로 되는지 식별하시오.

- 현재 자기 주위의 객체들과 클라스들을 여러개 들어 보시오. 그리고 그 객체들과 클라스들이 공통적으로 가지는 책임을 들어 보시오.

3 C++에 대한 소개(1)

이 장에서는 간단한 C++프로그람들을 보여 준다. 그리고 이 프로그람들을 보면서 C++의 기본조종구조들을 설명한다. 입출력명령문을 제외하고 C++언어의 조종구조는 C 언어에서와 같다.

3.1 간단한 C++프로그람

프로그람작성에 대한 대부분의 책들에서처럼 여기서도 사용자말단에 인사말을 출력하는 실례프로그람으로부터 시작한다.

```
# include  <iostream>

int main ( )
{
    std::cout << "Hello world " << " \n " ;
    return 0;
}
```

이 프로그람이 실행되였을 때 사용자말단에는 다음의 통보문이 표시된다.

```
Hello world
```

우의 실례프로그람에서 괄호 {와 }는 main함수의 본체를 막는데 쓰이였다.이 프로그람에는 식

```
std::cout << " Hello world "<< " \n " ;
```

이 있는데 이것은 현재출력흐름 std::cout 에 통보문 Hello world 를 쓰고 그다음 행바꾸기를 하라는것이다. 이 식은 현재출력흐름을 표시하는 std::cout 구체례에 통보문“Hello world”와 “\n”을 보내는것으로 생각할수 있다. 일반적으로 std::cout 는 말단으로 《귀착》된다. return 0;행은 결과 0 을 프로그람이 실행된 환경에 돌려준다. 약속에 따르면 돌림값 0 은 프로그람실행이 성공적이라는것을 의미한다. 그림 3-1 은 C++프로그람의 구조를 보여 준다.

주의: “\n”은 C++에서 행바꾸기문자를 표시한다. \문자는 그 다음문자가 해당한 의미를 가진다는것을 규정하는데 쓰인다. 앞의 경우에서는 행바꾸기를 표시한다. 모든 확장문자렬에 대한 서술은 부록 7 에서 제공한다.

#include < iostream >행은 C++언어의 부분이 아니다. 그것은 이 행을 iostream파일내용으로 교체하는 전처리지령이다. iostream파일에는 입출력처리에 대

한 정의가 들어 있다. 이 파일은 보통 콤퓨터의 체계등록부에 들어 있다. 이 행은 반드시 첫행에 있어야 한다.

```
# include <iostream>                    ← 프로그람에 입출력정의를 포함한다
int main( )
{                                        ← 《main 함수》프로그람의 입장점을 정의한다
std::cout << "Hello world " << " \n ";  ← 실행되는 명령문
return 0;                                ← 결과 0(성공)을 돌려 준다
}                                        ← main 함수를 끝낸다
```

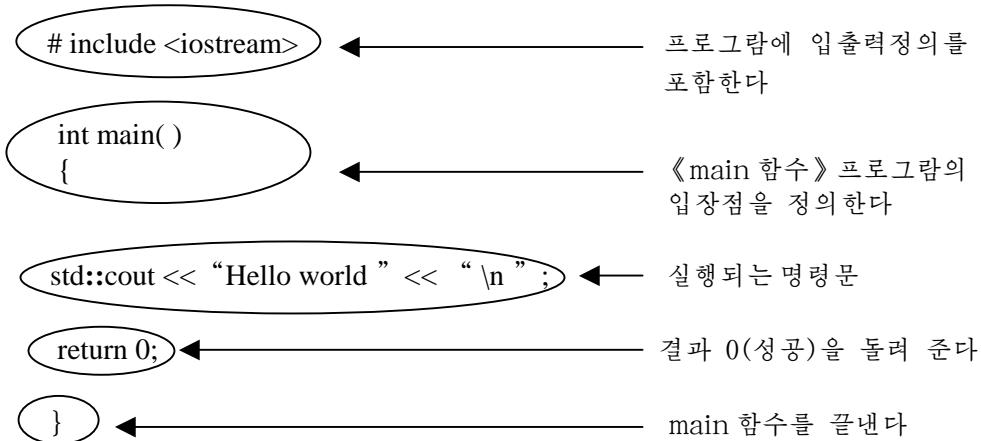



그림 3-1. C++프로그람의 구조

출력되는 항목들의 형들은 아래의 경우와 같이 바꿀수 있다. C++콤파일러는 해당한 출력형태를 알리기 위하여 항목형을 사용한다.

```
# include < iostream >
int main ( )
{
    std::cout << "The Sum of 1 + 2 + 3  is " << 1 + 2 + 3 << " \n ";
    return 0;
}
```

실행하면 다음의 결과가 나온다.

```
The Sum of 1 + 2 + 3  is  6
```

3.1.1 C++프로그람의 형식

C++프로그람에서는 프로그람을 이루는 개별적구성요소들을 구분할수 있는 형식으로 되여 있지 않아도 일 없다. 실례로 다음의 프로그람은 유효한 C++프로그람이다.

```
# include < iostream >
int main ( ) { std::cout << "Hello world " << " \n "; return 0; }
```

주의: 지령 # include는 그자체가 한행으로 되여야 하며 첫행에 놓여야 한다.
int와 main과 같은 자모문자단어들사이에는 적어도 한개의 공백문자가 있어서 서로 구별되여야 한다.

3.1.2 int main ( )

main(주프로그람)부분에는 실행프로그람의 결과로서 옹근수값을 돌린다는것을 의미하는 예약어 int 가 앞에 붙어 있다. 약속에 따르면 돌림값(returned value) 0은 프로그람의 실행이 성공이라는것을 의미한다. 프로그람은 사용자에게 오유통보문이 제시되여도 계속 실행될수 있다.

C++를 위한 ANSI 표준에서는 모든 C++콤파일러들이 주프로그람부분에서 다음과 같은 형식의 선언을 받아 들일것을 요구한다.

```
int main ( ) { }
int main ( int argc, char* argv [ ] )  { }
```

주의: 두번째 경우에서 main함수에 보내지는 값들에 대해서는 부록 11에서 서술한다.

물론 main 에 대한 다른 선언들도 콤파일러가 해석할수 있다.

3.1.3 설명문

C++에서 프로그람에 설명문을 덧붙이는데는 두가지 방법이 있다. 첫번째 방법을 보면 다음과 같다.

```
/* 설명문실례 */
```

여기서 설명문은 /* 과 */사이에 있다. 그러나 이 설명문은 일반적으로 다음과 같이 쓴다.

```
/*
*          이 프로그람은 C++콤파일체계에 대한 간단한 검사로서
*          통보문 Hello World 를 화면에 출력한다.
*/
```

주의: 설명문구획문자 /*과 */을 겹쳐 쓰지 않아도 된다.

두번째 방법은 다음과 같다.

```
// 행을 바꾸기전까지 설명문으로 된다.
```

여기서 설명문은 //으로 시작되며 행을 바꾸기전까지 설명문으로 인식한다.

주의: 일반적으로 프로그람내용을 인차 알아 볼수 없을 때 설명문을 달아 쉽게 리해하도록 한다.

3.2 보다 큰 C++프로그람

《수를 차례로 내려 가면서 세는》 프로그람을 아래에서 보여 준다. 이 프로그람에서는 조종흐름에 영향을 주는 여러가지 구조들이 소개되였다.

```
# include < iostream >

int main ( )
{
    int countdown = 10;                                    // 10 으로 설정
    while ( countdown > 0 )                                // 0 보다 클 동안
    {
        std::cout << countdown  << " \n " ;                //내리셈의 내용을 쓰기
        if ( countdown = = 3 )                             // 3 과 같다면
        {
            std::cout << "Ignition " << " \n " ;           // Ignition 출력
        }
        countdown--;                                       // countdown 을 1 감소
    }
    std::cout << "Blast Off " << " \n " ;                  // Blast off 를 출력
}
```

주의: countdown --는 변수내용을 1 개씩 감소시키는 C++략어이다. 이것은 countdown = countdown - 1;과 같다.

실행하면 결과는 다음과 같다.

```
10
9
8
7
6
5
4
3
Ignition
2
1
Blast Off
```

3.3 while에 의한 반복

다음의 명령문은 조건 countdown > 0 이 참이 아닐 때까지 { }사이의 코드를 반복실행한다.

```
while ( countdown > 0 )
{
        // 순환본체
}
```

주의: 조건의 앞뒤에는 ( )가 반드시 있어야 한다.
괄호 { }는 반복명령문이 여러개 있는 경우에만 쓴다. 그러나 일반적으로 순환경계를 보여 주기 위해 { }를 쓴다.

3.4 if에 의한 선택

다음의 명령문은 조건 countdown==3이 참이면 { }사이의 코드를 실행한다.

```
if ( countdown == 3 )
{
        // if 명령문본체
}
```

주의: 같기는 ==로 쓴다. 이것을 =로 쓰면 값주기가 아니므로 오유가 생길수 있다.

안같기는 !=로 쓴다. 실례로 countdown이 아직 령이 아닌가를 검사하기 위해 다음과 같이 쓸수 있다.

```
if ( countdown != 0 )
    std::cout << “Not  yet  zero ” << “ \n ” ;
```

주의: 조건이 참일 때 한개의 실행명령문이 있는 경우에는 { }를 쓰지 않는다.

3.4.1 if else

if 명령문에 else 부분을 추가할수도 있다. 즉

```
if ( countdown != 0 )
    std::cout << “Not yet zero” << “ \n ” ;
else
    std::cout << “Now zero” << “ \n ” ;
```

주의:

```
if (countdown != 0)
        std::cout << " Not yet zero" << '\n' ;
  else
        std::cout << " Now zero" << " \n " ;
```

반드시 있어야 한다.

else 전에 앞의 명령문들을 끝낸다는것으로서 ;을 반드시 표시하여야 한다.

3.5 다른 반복구조

3.5.1 for 명령문

C++에서 for명령문은 다음과 같다.

```
for ( int countdown = 10; countdown > 0; countdown -- )
{
        // for명령문본체
}
```

주의: for순환을 조종하는 변수 countdown은 ( )안에서 선언되여야 한다. 순환조종변수가 for명령문안에 선언되였을 때 그 유효범위는 순환본체이다. 26.4에 있는 유산콤파일러를 참고하시오.

이 실례에서 걸음은 10부터 1까지 감소한다. 이것은 다음의 while명령문과 같다.

```
{
   int countdown = 10;
   while ( countdown > 0 )
   {
      // 순환본체
      countdown --;
   }
}
```

주의: countdown = countdown - 1;은 C++략어로 countdown --이다. for 명령문에서 ;사이에 있는 임의의 부분은 생략될수 있다.

3.5.2 do while

일반적으로 순환이 적어도 한번은 실행될것을 요구하는 경우에 do while 명령문을 사용한다. 실례로 우의 for 명령문은 이 경우에 다음과 같이 쓸수 있다.

```
int countdown = 10;
do
{
    // 순환본체
    countdown--;
} while ( countdown > 0 );
```

3.6 다른 선택구조

3.6.1 switch

다음의 프로그람부분에서 if 명령문들은 복잡하게 결합되여 있다.

```
if ( number = = 1 )
    std::cout << "One" ;
else if ( number = = 2 )
    std::cout << "Two" ;
else if ( number = = 3 )
    std::cout << "Three" ;
else
    std::cout << "Not One , Two or Three" );
std::cout << " \n " ;
```

이것을 switch 명령문으로 다시 쓰면 다음과 같다.

```
switch ( number )
{
    case 1:
        std::cout << "One" ;
        break;

    case 2:
        std::cout << "Two" ;
        break;

    case 3:
        std::cout << "Three" ;
        break;

    default:
        std::cout << "Not One , Two or Three " ;
}
std::cout << " \n " ;
```

switch명령문에는 반드시 'break'를 서술하여야 하며 서술하지 않는 경우에는 조종이 다음 case표식(label)에로 넘어 간다. switch명령문에서 벗어 나려면 break명령문을 쓰면 된다.

주의: case표식은 옹근수기계단어로 결합될수 있는 값이든가 콤파일시에 상수로 될수 있는 값이여야 한다.
break가 없으면 실행은 case표식을 지나 다음 명령문을 계속 수행한다.

3.6.2 조건식명령문

다음의 식은 number 값에 따라 문자렬 《zero》 혹은 《not zero》를 준다.

```
( number = = 0 ? " zero " : " not zero " )
```

우의 조건식명령문은 number《형식》을 인쇄하기 위한데 사용될수 있다.

```
std::cout << "number is" << ( number = = 0 ? "zero" : "not zero" ) << " \n" ;
```

3.6.3 break 명령문

앞에서 설명한바와 같이 break명령문은 switch명령문을 벗어 나게 하는 조종에서 사용될수 있다. break명령문은 또한 순환구조들인 while, do while, for의 실행을 끝마치는데도 사용될수 있다.

주의: break의 위치를 잘못 정하면 오유가 발생될수 있다.

아래의 프로그람은 수자를 9부터 0까지 인쇄한다.

```
counter = 10;
while ( counter > 0 )
{
    counter--;
    std::cout << counter << " " ;
}
cout << " \n " ;
```

```
9 8 7 6 5 4 3 2 1 0
```

이때 break 명령문을 써서 counter 값이 3 과 같아 지는 경우 먼저 끝내게 할수 있다.

```
counter = 10;
while ( counter > 0 )
{
    counter--;
    if ( counter = = 3 )
        break;
    std::cout << counter <<  " " ;
}
cout <<  " \n " ;
```

결과 다음과 같이 9부터 4까지 수들만 인쇄한다.

```
9 8 7 6 5 4
```

3.6.4 continue 명령문

continue명령문은 일반적으로 사용하지 않는다. 이 명령문은 프로그람의 현재 실행경로를 무시하고 다음반복에로 넘어 가게 한다.

```
counter = 10;
while ( counter > 0 )
{
    counter--;
    if ( counter = = 3 )
        continue;
    std::cout << counter <<  " " ;
}
cout <<  " \n " ;
```

우의 코드를 실행하면 수자 3만은 인쇄되지 않는다.

```
9 8 7 6 5 4 2 1 0
```

continue는 do while과 for명령문들에서도 쓸수 있다. 이때 순환전반에서 현재 실행경로를 무시하고 다음반복에 대한 실행을 계속 시작한다.

주의: C++코드작성에서 안내선을 만들 때에는 대체로 case명령문의 탈퇴에 break만 쓰고 continue는 쓰지 않는다.

3.7 입력과 출력

C++에서 입력과 출력은 <<(삽입연산자)와 >>(추출연산자)를 사용하여 수행한다. 입출력함수들은 C++언어부분이 아니며 객체와 메쏘드들이 들어 있는 표준서고에 의해 제공된다. 일부 프로그람작성자들은 자체로 입출력체계를 작성하기도 한다. 그러나 표준입출력서고에서 제공된 객체들과 메쏘드들의 표준모임을 사용하는것이 일반적이다.

연산자 <<와 >>는 입출력처리전용이 아니지만 새 정의로서 다중정의된 C++연산자들이다. 다중정의(overloading)란 현재 있는 연산자나 함수에 또 다른 의미를 주는 처리이다.

3.7.1 출력

삽입연산자 <<는 현재출력흐름에 값을 보낼 때 쓴다.

```
std::cout << 42;
```

여기서는 std::cout(현재출력)와 결합된 흐름에 42를 쓴다. C++에서 std::cout는 보통 사용자말단에 자동적으로 접속되므로 또 다른 함수를 쓰지 않아도 된다.

결과출력형식은 다음과 같이 출력조작자를 사용하여 지정할수 있다.

```
std::cout << std::setprecision ( 2 );
```

우의 명령문은 모든 류점수들의 소수부자리를 두자리로 출력한다.

주의: 이 출력명령문은 아무런 물리적출력도 하지 않는다. 단지 모든 류점수들의 출력형식에 대하여 《두자리 소수부를 가진 소수로 표시》하도록 설정한다.

조작자 std::setprecision은 보통 다음과 같이 바꾸어 쓸수 있다.

```
std::cout << std::setiosflags ( std::ios::fixed );          // x.y형식으로 표시
std::cout << std::setiosflags ( std::ios::showpoint );      // 모든 자리수들을 표시
```

출력을 다른 출력흐름에 보낼수도 있다. 실례로

```
std::cerr << “Error in data” << “ \n ” ;
```

은 통보문을 오유출력흐름에 보낸다. 일반적으로 쓰는 std::cerr는 통보문을 사용자말단에 표시하라는것이다.

3.7.2 입력

추출연산자 >>는 프로그람에 자료를 입력한다.

```
int height;
std::cin >> height;
```

우의 코드는 옹근수값을 height 변수에 읽어 들인다.

입력에서 문제점은 모든 공백문자들(공백과 타브, 행바꾸기 등)이 기정값에 의해 무시된다는것이다. 실례로 본문

```
In C++ the operator:
    <<    is used for output
    >>    is used for input
```

을 입력흐름 std:cin 으로 읽어 들여 출력흐름 cout 에 한 문자씩 쓴다면 출력은 다음과 같다.

```
InC++theoperator:<<isusedforoutput>>isusedforinput
```

이것을 해결하기 위해서는 공백무시요청항목을 비설정으로 하는 다음의 조작자를 사용하면 된다.

```
std::cin >> std::resetiosflags ( std::ios::skipws );
```

주의: 입출력조작자를 사용하기 위하여서는 머리부파일을 프로그람에 넣어야 한다.

```
# include <iomanip>
```

부록 1 에서는 기본입출력함수에 대하여 구체적으로 서술한다.

3.7.3 이름공간 std

많은 사람들이 참가하여 만든 큰 프로그람들에서는 항상 각이한 프로그람작성자들이 같은 이름으로 된 항목을 쓸수 있는 위험이 따른다. 이것을 방지하거나 표준서고에 들어 있는 이름들과의 충돌을 피하기 위하여 이름공간을 사용한다. 이름공간(namespace)은 이름 붙은 항목들에 대하여 교갑화를 제공한다.

표준서고객체들과 함수들 등은 모두 이름공간 std 안에 들어 있다. 이름공간 std 에서 항목을 호출하려면 서고항목이름앞에 std::를 붙인다. 그때문에 객체 cout 와 cin 은 항상 std::cout 와 std::cin 으로 호출된다. 이름공간에 대해서는 13 장에서 보다 구체적으로 서술한다.

서고항목	설　명
std ::cout	이름공간 std 에 들어 있는 현재출력흐름객체.
std::ios::skipws	항목 skipws .ios 는 이름공간 ios 에 들어 있으나 이름공간 std 안에는 들어 있지 않다(이것은 정확히 그렇다고 말할수는 없지만 현재는 그렇다고 가정하자).

3.8 반점연산자

다음의 프로그람은 입력을 한 문자씩 출력에 복사하기 위한 간단한 쏘프트웨어도구로 쓸수 있다. C++에서 cout 는 표준출력흐름이고 std::cin 은 표준입력흐름이다.

,연산자는 두번째 인수의 결과를 넘겨 준다. 아래의 프로그람에 있는 식

```
std::cin >> ch , !std::cin.eof ( )
```

에서 ,연산자는 두번째 인수의 결과를 넘겨 주기 위하여 사용된다. 이것은 명령문 std::cin>>ch; 를 반복하여 사용하는것을 피하게 한다.

주의: ,연산자를 쓴 두 연산수의 형들은 아래의 프로그람에서 사용한것처럼 서로 다를수 있다.

```
# include <iostream>
# include <iomanip>

int main ( )
{
    char ch;                                                    // ch 선언
    std::cin >> std::resetiosflags ( std::ios::skipws );
    while ( std::cin >> ch , !std::cin.eof ( ) )               // 파일끝이 아닌 동안
    {
        std::cout << ch;                                       // ch 에 읽어 들인것을 출력
    }
    return 0;
}
```

주의: std::cin >>std ::restiosflags (std::ios ::skipws);
이 행에 의하여 입력흐름이 공백문자(white space)를 포함한 모든 문자들을 내여 준다. 공백문자에는 공간(space)과 타브, 행바꾸기 등이 있다.

우의 프로그람은 파일의 내용을 인쇄하는 간단한 쏘프트웨어도구로서 UNIX 체계에서 쓸수 있다. 만일 콤파일된 프로그람 simple_cat 를 실행하면 다음과 같이 Text 파일내용을 인쇄한다.

```
% simple_cat < Text
C++ was initially designed by Bjarne Stroustrup between
1980-1983 and was initially invented to help write event
driven simulations.
As the name implies it is the next increment from C
combining the concept of classes from Simula67 together
with features from BCPL the original inspiration for C.
In may ways C++ is a superset of C.
```

3.9 자체평가

- 지금까지 본 C++와 자기가 알고 있는 다른 콤퓨터프로그람작성언어와의 기본적인 차이점은 무엇인가?
- else부분이 없는 간단한 if와 while구조를 함께 사용하여 if else구조 혹은 do while구조로 표현할수 있는것들을 모두 서술할수 있는가?
- 식 1 == 2의 결과는 무엇인가? 무효한 식 1 = 2와 무엇이 차이나는가?
- 다음의 행이 실행되면 무엇이 출력되는가?

```
std::cout << “C++ was designed by Bjarne Stroustrup” << “ \n ” ;
```

- switch명령문을 같은 기능을 수행하는 if else명령문으로 교체할수 있는가? 변환이 된다면 그 리유를 설명하시오.
- C++에서 break와 continue명령문들이 필요한가?
- C++는 왜 두개의 순환구조 while과 do while을 가지고 있는가?
- 프로그람에서 언제 설명문을 사용하는가?
- 다음의 프로그람은 무엇을 수행하며 일반적으로 이렇게 서술하는것이 옳은가?

```
# include <iostream>
int main ( )
{
    int sum = 0;
    for ( int count = 1; count < 100; count++ )
    {
        sum = sum + count;
        if ( sum > 1000 )
        {
            std::cout << count << “ \n ” ;
            count = 100;
        }
    }
}
```

- =와 ==의 차이점은 무엇인가?

3.10 련 습

다음의 프로그람을 작성하시오.

- 수
  while 순환을 사용하여 수 32 부터 126 까지 쓰는 프로그람.
- 구구표
  5 계단구구표를 인쇄하는 프로그람. 출력은 다음과 같은 형식으로 하시오.

  5 * 1 = 5
  5 * 2 = 10
  5 * 3 = 15
  . . .
  5 * 9= 45

- 일반적인 경우의 구구표
  임의의 계단에 해당한 구구표를 인쇄하는 프로그람. 사용자에게 어느 계단구구표를 요구하는지 물어 보는 명령문이 있어야 할것이다.

  참고 : num에 10진수를 읽어 들이는 코드는 다음과 같다.

```
int num ; std::cin >> num ;
```

- 수렬
  수렬 1 1 2 3 5 8 13 … 에서 마지막항목이 10000보다 커질 때까지 항목을 인쇄하는 프로그람.
- 순위
  세 수를 읽고 커지는 차례로 인쇄하는 프로그람.

4　C++에 대한 소개(2)

이 장에서는 C++에서 서술할수 있는 간단한 자료항목들중에서 그 일부를 서술한다.

4.1　소개

```
int  countdown;
```

지금까지의 실례에서는 객체의 형을 int 혹은 char형으로만 선언하였다. int형의 객체가 가질수 있는 정확한 표현이나 《값범위》는 C++의 실현에서 차이날수 있다. 그 결과는 콤퓨터가 쓰는 기계단어의 byte수에 따른다.

주의: 표준머리부파일 <limits>와 <float.h>에는 해당한 체계에서 실현되는 자료형에 대한 최대값과 최소값이 들어 있다.

C++에서는 변수가 선언될 때 동시에 초기화되게 할수 있다. 즉

```
int countdown = 10;
```

4.2　자료항목의 선언

자료항목들은 프로그람의 각이한 위치에서 선언될수 있다. 그 위치는 실행명령문들과 함께 있을수도 있는데 일반적으로는 함수의 시작이라든가 혹은 시작기호 { 다음에 선언된다.

```
# include <iostream>
int main ( )
{
    int green_cars = 23;                      // 자료항목이 선언될 때
    int red_cars    = 20;                     // 초기값도 함께 줄수 있다
    std::cout << “Number of green cars = ” << green_cars << “ \n ” ;
    std::cout << “Number of red cars     = ” << red_cars << “ \n ” ;
    int total = green_cars + red_cars;
    std::cout << “Total number of cars  = ” << total << “ \n ” ;
    return 0;
}
```

4.2.1 식별자

C++식별자(변수이름 혹은 함수이름 등)는 대문자와 소문자, 수자, 밑선(_)으로 이루어 진다. 식별자(identifier)의 최대길이에는 제한이 없으나 반드시 영문자로부터 시작해야 한다.

4.3 C++의 기본형

내장된 자료형이나 기본형들은 다음과 같다.

형	다른 표기법	해 설
char		문자를 보유한다.
unsigned char		부호 없는 문자형
signed char		부호 있는 문자형
bool		론리값
wchar_t		배길이문자형
int	signed int signed	옹근수를 보유하는 기계단어
unsigned int	unsigned	부호 없는 옹근수
short	signed short int signed short short int	짧은 옹근수
unsigned short	unsigned short int	부호 없는 짧은 옹근수
long	long int	긴 옹근수
unsigned long	unsigned long int	부호 없는 긴 옹근수
float		류점수
double		배정확도류점수
long double		긴배정확도류점수

주의: float, double과 long double은 류점수로 기억된다. 나머지기본형들은 정확한 량으로 기억된다.
부호 없는 형들은 부호 있는 형보다 더 큰 정수를 보관할수 있다.
이 자료형으로 선언된 항목들의 정확도는 사용되는 콤파일러나 콤퓨터에 따라 다르다.
char표시에는 부호가 있을수도 있고 없을수도 있다. 따라서 실현프로그람이 char의 산수적표현을 쓸수 있게 한다.

4.4 사용자정의형

C++프로그람을 작성하는데서 문제로 되는것은 수의 정확도를 지정할수 없는것이다. 더우기 이 언어를 실현할 때마다 기본형들의 정확도를 변화시켜야 하는 경우도 제기된다. 다음의 코드부분을 고찰하자.

```
int big_number = 100000;
```

일부 콤퓨터들에서 이 선언은 오유로 된다. 왜냐하면 int가 2byte길이로만 정의되기때문이다. int의 2byte로는 -32768부터 32767까지의 범위만 표시할수 있다. 이 문제를 해결하기 위하여 C++에서는 현재자료형에 의하여 큰 수를 표시하는 새로운 자료형을 정의할수 있게 한다. 이것은 다음과 같은 사용자정의형(typedef)명령문으로 수행할수 있다.

```
typedef int Interger;
Integer big_number;
```

여기서 새로운 형 Integer는 int와 같다. int가 2byte길이이고 long int가 4byte길이를 가지는 콤퓨터에서 typedef명령문은 다음과 같이 변화시킬수 있다.

```
typedef long Interger;
```

일반적으로 한개의 머리부파일에 typedef명령문을 써서 다른 콤퓨터에서 실행시킬 때 프로그람에 대한 변환이 적어 지게 해야 한다.

주의: typedef는 새로운 형이 아니라 현재 있는 형선언에 대한 별명을 소개한다.

4.5 float와 double, long double형

int형구체례에는 수가 옹근값으로 들어 있다. 어떤 문제를 푸는데서 조종되는 수자들은 옹근값이 아닐수도 있다. 실례로 사람의 무게는 80.45kg이다. float형의 구체례는 소수가 들어 있을수 있는 변수이다. 따라서 프로그람에서 사람의 무게는 다음과 같이 선언된 weight변수에 들어 갈수 있다.

```
float weight = 80.45;
```

float형의 구체례는 류점수로서 실현된다. 류점수는 값이 소수로 들어 있다. 류점수는 많은 경우에 프로그람작성자가 기억시키려는 옹근값의 근사값으로 될수 있다. 실례로 1/3은 0.333…33으로 들어 있을것이다. 다음의 표는 수들이 6자리의 소수부를 가진 류점수형식으로 어떻게 효과적으로 기억되는가를 보여 준다.

수	과학적인 표기법	류점형식
80.45	0.8045×10^{2}	+804500 + 02
0.008045	0.8045×10^{-2}	+804500 － 2
0.333333	0.333333×10^{0}	+333333 + 00

주의: 실지로 류점수는 2진수로 들어 있다.

류점수를 사용하는 기본목적은 수를 그의 진짜 값에 대한 근사값으로 보유하자는것이다. 류점수를 사용한 계산은 일반적으로 정확한 값에 대한 근사값만 줄것이다. 많은 경우에 이 근사값은 문제로 제기되지 않으나 화페량을 표시하는 경우에는 문제가 있다. C++의 정의에서 double은 float보다 크거나 같은 정확도를 가지며 long double은 double보다 크거나 같은 정확도를 가진다.

4.6 상수선언

프로그람을 쉽게 보기 위하여 상수들에 기호이름을 줄수 있다. 상수(const)선언은 일반변수와 아주 비슷하며 단지 차이점이라는것은 후에 거기에 어떤 값주기도 할수 없다는것이다.

```
const int           MAX = 10;
const long double   PI = 3.14159265358979323846264338327 9L;
```

주의: 일반적으로 프로그람작성에서는 0 혹은 1을 제외한 임의의 값에 대하여 상수정의를 사용한다.
상수항목들과 변수항목들을 쉽게 구별할수 있는 점은 상수항목들은 대문자로 표시된다는것이다.
long double상수를 가리키는 수의 마감에는 L을 붙인다.
부록 9에서는 C++프로그람에서 선언될수 있는 각이한 문자형들을 보여 준다.

4.7 렬거형

렬거형(enumeration)변수는 기정값을 가진다. 실례로 아래의 프로그람은 붉은색 혹은 푸른색, 풀색값만 가질수 있는 Colour라는 이름의 새로운 형(렬거형)을 창조한다. 콤파일시 어떤 다른 값이 Colour형변수에 값주기되였다면 오유가 발생된다.

```
enum Colour { RED, BLUE, GREEN };            // 렬거형
Colour car;
car = RED;
switch ( car )
{
   case RED:
      std::cout << “Car is red”;
```

```
        break;
    case BLUE:
        std::cout << “Car is blue”;
        break;
    case GREEN:
        std::cout << “Car is green”;
        break;
}
```

주의: 콤파일러는 RED를 0으로, BLUE는 1…로 표시한다. 그러나 enum colour {RED=2, BLUE=4, GREEN}로 초기화하여 이 값들을 바꿀수 있다. GREEN은 그 다음값 즉 5를 가진다. 일반적으로 프로그람작성에서는 렬거형이 기정값을 가질 때는 초기값을 따로 설정하지 않아도 된다.
매 case 표식의 마감에 break 명령문을 써서 switch 명령문을 벗어 난다.

4.8 산수연산자

C++에서 산수연산자들은 다음과 같다.

+	더하기
–	덜기
*	곱하기
/	나누기
%	모듈러연산, 나머지연산

주의: 나누기:연산수들이 옹근수형이면 옹근수나누기를 한다. 그러나 연산수중 한개가 류점수이면 류점수나누기를 한다.
모듈러연산:연산수들은 옹근수여야 한다.

실례로 프로그람

```
# include <iostream>

int main ( )
{
    std::cout << 5 / 2    <<  “ \n ”;                    // 옹근수나누기
    std::cout << 5 / 2.0 <<  “ \n ”;                    // 류점수나누기(5.0/2.0)
    std::cout << 5 % 2  <<  “ \n ”;                    // 모듈러연산
    return 0;
}
```

을 콤파일하고 실행하면 결과는 다음과 같다.

```
2
2.5
1
```

4.8.1 단항산수연산자

–	부수
+	정수

단항산수연산자에는 –와 +가 있다.

4.9 관계연산자

==	갈기
!=	안갈기
<	작기
>	크기
<=	작거나 갈기
>=	크거나 갈기

주의: 다시 강조하지만 갈기는 ==로 쓰며 =는 갈기가 아니라 값주기연산자이다. 기본자료형들만이 관계연산자들에 의하여 비교될수 있다. 특히 두개의 문자렬을 비교하여 갈기비교 혹은 순서맞추기를 한다면 결과가 나오지 않는다. 그리유에 대해서는 8.12.3 항목에서 설명한다.

실례로 다음과 같은 식을 쓸수 있다.

```
std::cout << ( temperature > -10 ? “Maybe warm” : “Very cold” );
```

4.10 론리연산자

&&	론리적
||	론리합

론리연산자에는 &&와 ||가 있다.

론리연산자들로 여러개의 조건식을 구성할수 있다. 즉

```
if ( year = =  2004  &&  month  = =  2 )
{
   // 29 일간
 }
```

부록 6에는 모든 연산자들에 대한 우선순위목록이 들어 있다. C++에서 &&와 ||는 관계연산자보다 더 낮은 우선순위를 가진다.

주의: 조건식평가는 왼쪽에서 오른쪽으로 가면서 평가되며 요구하는 결과가 성립되면 곧 평가를 끝낸다. 이 불완전한 평가는 모든 조건부분을 실행해야 할 때 문제를 일으킬수 있다. 8.12.1 항목에서는 식의 불완전평가를 피하는 방법에 대하여 서술한다.

4.10.1 단항론리연산자

!	부정

이것은 론리식이나 론리값의 반전을 넘겨 준다.

실례는 다음과 같다.

```
std::cout << ( !my_birthday ? “normal day” : “My birthday” );
```

4.11 론리형

론리(bool)형은 진리값이 들어 있는 변수를 선언한다. 실례로 론리변수 newyear 는 다음의 코드에서 사용된다.

```
bool newyear = day = = 1 && month = = 1;

if (newyear)
{
    std::cout << “A Happy New Year” << “ \n ” ;
}
```

4.11.1 론리형의 대치

처음에 C++에는 론리형이 없었다. 론리값조종을 위하여 대신 사용되는 int값은 다음과 같다.

진리값	int 값에 의한 표시
True	0 을 제외한 임의의 값
False	0

진리값표현에 int 값을 리용하는 방법은 원천코드를 효과적으로 작성하게 한다. 실례로 한행에 《－》문자를 40 개 쓰는 코드는 다음과 같다.

```
int count = 40;
while ( count )
{
    count--; std::cout << “-” ;
}
```

진리값들을 처리하는 우의 방법과 호환성을 맞추기 위하여 bool 형구체례는 다음의 int 형값들로 형변환된다.

선언	std::cout << (int) res;
bool res = true;	1
bool res = false;	0

4.12 비트연산자

비트연산자들은 옹근수량의 연산에 사용된다. 이 연산자들은 일부 경우에만 사용된다.

&	비트적
|	비트합
ˆ	배타적비트합
<<	형식 <<n : 2진 n자리 왼쪽 밀기형식
>>	형식 <<n : 2진 n자리 오른쪽 밀기형식

주의: 앞의 실례들에서 연산자 <<와 >>은 지금까지 입출력용으로 써왔다. 이 연산자를 입출력용 혹은 밀기용으로 쓸수 있는것은 C++가 연산자들을 새로운 이름으로 다중정의할수 있게 하기때문이다. 이에 대해서는 15장에서 구체적으로 서술한다.

실례로 다음의 프로그람

```
# include <iostream>

int main ( )
{
    for ( int i = 1; i <= 4; i++ )
    {
        std::cout << ( 32 >> i ) << “\t” << ( 32 << i ) << “ \n ” ;
    }
    return 0;
}
```

을 콤파일하고 실행하면 결과는 다음과 같다.

```
16   64
8    128
4    256
2    512
```

주의: C++에서는 타브(tab)기호를 “\t”로 표시한다. 부록 7에 문자렬 혹은 문자상수에 포함될수 있는 확장문자렬을 주었다. <<은 출력삽입연산자와 왼쪽밀기연산자로 사용할수 있다.

4.12.1 론리연산자 |와 &의 사용에서 주의점

|과 &는 론리연산자로 사용할수 있지만 불완전평가에서는 쓰지 말아야 한다.

식	평가결과

if (temperature > 20 & sunny ( ))
{
work_outside ( );
}
	식 temperature > 20 과 함수 sunny( )는 temperature가 20이상인 경우에도 평가된다.

4.12.2 C++에서 단항비트연산자

~	비트부정

이 연산자는 비트단위로 단어의 모든 0을 1로, 1을 0으로 바꾼다.

4.13 sizeof 연산자

sizeof 연산자들은 C++자료형의 크기를 준다. 아래의 실례코드는 내장된 일부 자료형의 byte 크기를 인쇄한다.

```
# include <iostream>

int main ( )
{
    std::cout << “Sizes  of   types  ” << “\n ”;
    std::cout << “char          ” << sizeof ( char )          << “\n ”;
    std::cout << “short         ” << sizeof ( short )         << “\n ”;
    std::cout << “int           ” << sizeof ( int )           << “\n ”;
    std::cout << “bool          ” << sizeof ( bool )          << “\n ”;
    std::cout << “long          ” << sizeof ( long )          << “\n ”;
    std::cout << “float         ” << sizeof ( float )         << “\n ”;
    std::cout << “double        ” << sizeof ( double )        << “\n ”;
    std::cout << “long double ” << sizeof ( long double )   << “\n ”;
    return 0;
}
```

두대의 각이한 콤퓨터들로 실행할 때 결과는 다음과 같다.

Sun sparc 형	PC 80x86 DOS 형
Sizes of types char 1 short 2 int 4 bool 1 long 4 float 4 double 8 long double 8	Sizes of types char 1 short 2 int 2 bool 1 long 4 float 4 double 8 long double 10

주의: 한개의 프로그람이 여러 콤퓨터들에서 실행되여야 하는것이라면 이 정보는 치명적이다.

4.14 변수의 형변환

많은 언어들에서와 같이 C++는 산수연산을 수행할 때 형변환(promotion)에 대해 엄격한 규칙을 가지고 있다. 그러나 C++는 연산이 진행되는 많은 부분들에서 형변환을 한다. 이때 잘못하면 오유를 발생시킬수 있다. 다행히도 이 형변환은 아주 직관적이다.

표준산수연산에서 변수내용은 필요에 따라 연산이 수행되는 형식에 맞추어 형변환된다. 이 처리는 여러 단계로 이루어 진다.

《연산수 1》《연산자》《연산수 2》를 포함한 식에서

- 만일 연산수 1 혹은 연산수 2 가

long double	인 경우 다른 연산수는	long double	로 변환된다
double	인 경우 다른 연산수는	double	로 변환된다
float	인 경우 다른 연산수는	float	로 변환된다

- 그렇지 않으면 옹근수형변환은 두 연산수에 대하여 다음과 같이 진행된다.
  - ✧ 형(char 혹은 signed char, unsigned char, short int, unsigned short int)의 모든 값들이 int 로 표시될수 있다면 그 형의 구체례는 int 로 변환된다. 그렇지 않으면 그 형의 구체례는 unsigned int 로 변환된다.
  - ✧ 형(wchar_t 혹은 렬거형)구체례에서 값은 모든 값들을 표시할수 있는 int, unsigned int, long, unsigned long 형중 높은 지위의 연산수형으로 변환된다.
  - ✧ 론리형의 구체례에서 false 는 0, true 는 1 로 하는 int 형구체례로 변환된다.

실례로 char 의 구체례는 int 의 구체례로 형변환된다.

- 만일 연산수1 혹은 연산수2가운데서 어느 한 연산수가

unsigned long 인 경우 다른 연산수는 unsigned long 으로 변환된다

- 한 연산수가 long int이고 다른 연산수가 unsigned long인 경우는 다음과 같다.

  long int가 unsigned int 로 표시될수 있다면 unsigned int는 long int로 변환된다.
  long int가 unsigned int로 표시될수 없다면 두 연산자들은 unsigned long int로 변환된다.

- 만일 연산수1 혹은 연산수2가운데서 어느 한 연산수가

long	인 경우 다른 연산수는	long	으로 변환된다
unsigned	인 경우 다른 연산수는	unsigned	로 변환된다

두 연산수들로 이루어 진 식에서 변수들의 형변환은 다음의 3 단계를 거쳐 진행될수 있다.

첫째 단계 어떤 연산수들이 다음의 형이라면	**둘째 단계** 그러면 그 연산수들중 아래의 두개 형가운데서 낮은 지위에 있는 것이 형변환된다.	**셋째 단계** 만일 두 연산수들이 서로 다른 지위에 있다면 낮은 지위에 있는 연산수가 높은 지위에 있는 연산수와 같은 형으로 변환된다.	
char			낮은 지위
signed char			
unsigned char			
short int			
unsigned short int			
wchar_t (주의를 볼것)			
bool			
	int		
	unsigned int		
		long int	
		unsigned long int	
		float	
		double	
		long double	높은 지위

주의: 형변환사용에서 중간형변환이라는것은 없다. short int 와 double 을 포함하는 식에서 short int 는 double 로 형변환된다.
wchar_t 의 구체례는 wchar_t 형의 모든 값이 들어 갈수 있는 형 int, unsigned int, long, unsigned long 들중 높은 지위의 구체례로 변환된다.
우와 같이 지적된 특수경우이면 연산수들은 long int 이거나 unsigned int 이다.

례를 들어 다음의 코드토막을 보자 .

```
short int units = 5;
float price      = 123.45;
std::cout << units  *  price;
```

식 unit * price 에서 옹근수형변환은 먼저 short int 를 int 로 형변환한다. 따라서 두 연산수들은 이 경우 float 로서 같은 지위로 되였다.

4.15 강제형변환

C++에서 한개의 자료형은 강제형변환에 의해 다른 형으로 변환될수 있다. 실례로 수가 다음과 같이 문자로 변환될수 있다.

```
std::cout << (char) 65;
```

이 코드를 실행하면 문자 A 가 인쇄된다. 이것은 물론 콤퓨터가 ASCII 문자모임에 기초하여 작업하고 있다고 가정한 경우이다.

주의: C++는 프로그람작성자들을 위하여 자동적으로 일부 형들을 변환한다. 그러나 우의 실례에서 출력루틴들은 자기의 형값을 어떻게 출력할것인가를 결정하여야 한다. 이때 강제형변환이 필요하다.

다음과 같은 함수적표기법도 가능하다.

```
std::cout << char (65) ;
```

주의: 함수적표기법이 또 다른 구조(클라스항목에 대한 구축자)를 서술하는데 사용되여 그 표기법을 사용할수 없는 경우가 생길수도 있다. 그러므로 형을 괄호로 막은 첫번째 표기법을 사용하는것이 제일 좋다. 그것은 《함수적표기법》이 구축자의 호출을 가리킬수 있기때문이다.

4.16 연산의 간략법

C++에는 변수에 량을 더하거나 더는 간략법(shortcut)들이 있다. 많이 사용되는것은 증가연산자와 감소연산자들인 ++와 --이다. 이 연산자들은 쓰는 위치에 따라 결과가 달라 진다.

연산자	값의 앞에 있는 경우 (실례로 ++cost)	값의 뒤에 있는 경우 (실례로 cost++)
++	값을 한개 증가시키고 결과 넘기기	값을 넘기고 그 값을 한개 증가
--	값을 한개 감소시키고 결과 넘기기	값을 넘기고 그 값을 한개 감소

실례로 cost 가 아래의 모든 경우에 대하여 초기값 10 을 가진다면 다음과 같다.

C++식	식까지의 결과값	cost 결과값
++cost	11	11
cost++	10	11
--cost	9	9
cost--	10	9

주의: 이 연산자들은 산수형 또는 지적자형에서도 쓰일수 있다. 지적자에 대하여서는 17 장에서 충분히 서술한다.

4.16.1 다른 간략법

간단히 표시할수 있는 또 다른 일반적인 구조는 다음과 같은 식이다.

```
항목 = 항목 연산자 식;
```

여기서 연산자는 +, -, *, / 등이다. 이것을 다음과 같이 생략할수 있다.

```
항목 연산자 = 식;
```

실례로 다음과 같은것들을 들수 있다.

완전한 형식	간략형식
money = money + 100;	money += 100;
reduce_price = reduce_price / 2;	reduce _price /= 2;

4.17 식

C++에는 값주기명령문(assignment statement)은 없고 식명령문(expression statement)이 있다. 그 차이점은 식명령문이 값주기명령문처럼 보이지만 사실은 식이라는것이며 그 식이 결과를 넘겨 준다. 따라서 C++프로그람은 다음과 같이 쓸수 있다.

```
int a , b , c ;
a = b = c = 0 ;
```

이것은 a, b, c 의 값을 0 으로 설정하라는것이다. 이것은 =(값주기)가 대입값을 넘겨 주기때문이다.

그리고 다음과 같은 식도 역시 타당한 식이다.

```
i ;
```

이 명령문은 i 의 내용을 넘겨 주고 곧 없어 진다.

주의: 우의 명령문은 코드실현을 할 때 빠짐오유로 보일수 있다.

4.18 연산자의 개요

C++에서 자료형에 쓰이는 연산들은 아래와 같다. 이 표에서 int 에는 short int 와 long int , signed int , unsigned int 도 포함되며 float 에는 double float 와 long float 도 포함된다.

		허락표시		
연산자	**설명**	**char**	int	float
+	더하기	√ ?	√	√
−	덜기	√ ?	√	√
/	나누기	√ ??	√	√
*	곱하기	√ ??	√	√
%	모듈러연산, 나머지연산	√ ??	√	

연산자	설명	허락표시		
		char	int	float
<<	왼쪽밀기	√	√	
>>	오른쪽밀기	√	√	
&	비트합	√	√	
|	비트적	√	√	
ˆ	배타적비트합	√	√	
&&	론리합	√	√	
||	론리적	√	√	

√ 허락된다.
√ ? 허락되지만 주의하시오. 그것은 요구대로 되지 않을수 있기때문이다.
√ ?? 허락되지만 꼭 그렇게 해야 하겠는가?

주의: 표에서 int 와 float 는 모든 int 형들과 float 형들, 실례로 unsigned int 와 long double 을 표시한다.

4.19 자체평가

- C++에서 변수들을 어떻게 선언하며 그 선언은 어디에서 해야 하는가?
- C++에서 옹근수의 정확도란 무엇인가?
- 한개의 표준자료형구체례가 그밖의 표준자료형구체례로 어떻게 형변환되는가? 어떤 제한과 금지조건들이 있는가?
- C++에 들어 있는 내장자료형을 표로 작성하시오. sizeof 를 사용하여 모든 내장자료형들사이의 관계를 줄수 있는가?
- 다음의 두행들사이의 차이점은 무엇인가?

```
int i=10;  std::cout << i ++ ;
int i=10;  std::cout << ++ i ;
```

- char c ; int i ; double d ; unsigned int u ;
  라고 선언된 경우 다음식의 결과형은 무엇인가?

```
c + i ;            d + u ;              i + d ;
c + c ;            i + i ;              u + u ;
( char ) i + c ;   ( double ) i + i ;   c + u ;
```

- 나누기연산자는 언제 옹근수를 넘겨 주며 언제 류점수를 넘겨 주는가?
- 다음의 코드가 실행될 때 무엇이 인쇄되는가?

```
int main ( )
{
  int number = 512;
  number << = 4;
  std::cout << number;
  std::cout << "\t" << ( 1 & 3 ) << "\t" << ( 5 % ( 4 >> 1 ) );
  return 0;
}
```

4.20 련 습

다음의 프로그람을 작성하시오.

- 문자코드
  수 32 부터 126 에 해당하는 ASCII 문자를 써내는 프로그람.
  출력을 다음과 같이 하시오.

  ASCII char A represented internally by internal code 65.
  ASCII char B represented internally by internal code 66.

- 문자계수
  표준입력으로부터 들어 오는 문자들의 개수를 세는 프로그람. 이 프로그람은 본문파일에 있는 문자의 개수를 세는 쏘프트웨어도구로서 사용될수 있다.

- 대문자
  문장의 첫 문자를 대문자로 표시하는 프로그람.

- 씨수
  1 부터 1000 사이의 수가운데서 모든 씨수를 인쇄하는 프로그람.

- 완전수
  1 과 1000 사이의 모든 완전수들을 인쇄하는 프로그람.

어떤 수의 약수들을 모두 더한 합이 그 수와 같을 때 그 수를 완전수라고 말한다. 실례로 6 은 완전수(약수들은 1,2,3)이고 28 도 완전수(약수들은 1, 2, 4, 7, 14)이다.

- 어떤 수의 2 진수 1 비트들의 개수
  옹근수를 입력하고 그의 2 진수표시에서 1 이 몇개 들어 있는지 인쇄하는 프로그람. 실례로 수자 5 가 입력되면 결과는 2 로 되고 8 이 입력되면 결과는 1 로 된다.

- 부호화
  간단한 부호화알고리듬은 비트형식으로 된 본문통보문의 모든 문자들에 대하여 배타적론리합(배타적론리합연산자 ^)을 적용하는것이다. 그다음 그 부호화된 본문에 우의 과정을 반복하여 원래통보문을 회복할수 있다. 이 부호화, 복호화를 수행하는 프로그람을 작성하시오. 실례로

'A'	01000001	부호화	00110001
열쇠	01110000	열쇠	01110000
배타적론리합	------------	배타적론리합	------------
부호화	00110001	복호화	01000001

참고: 2 진자료를 피하기 위하여 7bit 열쇠를 사용한다.

5 클라스

이 장에서는 클라스에 대하여 보기로 한다. 클라스는 한개의 단위안에 서로 대화하는 코드와 자료를 교갑화하는 품위 있는 방법이다. 이 단일단위는 객체라고 하는 클라스의 구체례를 창조하는데 리용된다.

5.1 소개

우리가 살고 있는 세계에는 매일 더 헐하고 더 즐거운 생활을 누리게 하는 대중수단들과 기구들이 많다. 례를 들어 텔레비죤은 거의 모든 사람들이 본다. 그런데 《그 함》안에서 정확히 무엇이 진행되는지 알고 있는 사람은 거의 없다. 마찬가지로 수만명의 운전수들이 차에 대하여 구체적으로 알고 있지 않아도 능동적으로 몰수는 있다. 사람들은 대체로 승용차에 대하여 그림 5-1에서처럼 알고 있다.

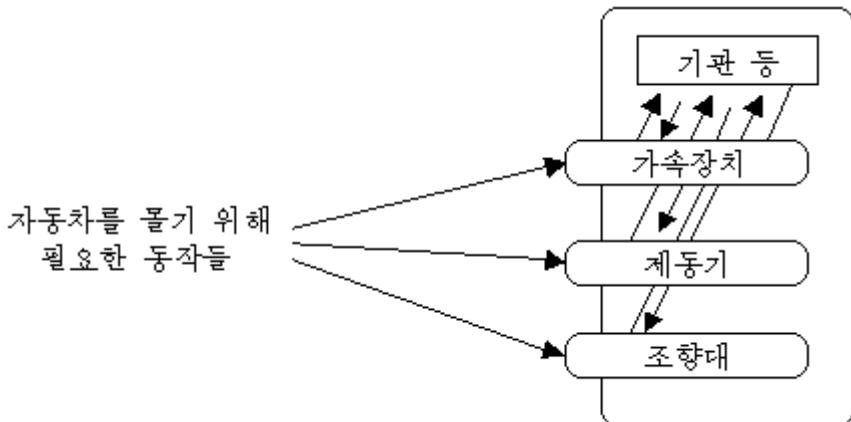



그림 5-1. 자동차를 몰기 위한 기본적인 리해

승용차내부가 하는 동작에 대하여서는 매일 차를 모는 운전수들에게 있어서 그리 중요치 않다.

본질적으로 세계에는 매 항목들을 능률적으로 사용하도록 하기 위하여 일반사람용대면부를 가지는 객체들이 널리 퍼져 있다. 사람들은 때때로 그 대면부들이 사용하기에는 비능률적이고 까다롭다고 비난한다. 아직까지 많은 경우에 사람들은 다른 수단에 의거해서 과제를 수행하는것보다 자기들이 흔히 보는 객체를 쓰는것을 더 좋아 한다.

쏘프트웨어세계에서도 사용자나 프로그람작성자들이 그 객체의 내용을 모르고도 효과적으로 사용할수 있게 하는 객체들이 있다. 아주 간단한 C++프로그람을 보면 류점수들이 들어 가는 변수들을 선언한 다음 그 값들로 더하기, 곱하기 등의 산수연산을 한다. 그러나 대부분의 프로그람작성자들은 이 연산들이 어떻게 집행되는지 그

정확한 세부내용에 대하여서는 모른다. 단지 그 언어에서 제공되는 대면부를 받아들여 산수연산을 한다.

얼핏 보면 프로그람작성자들에게 풍부한 자료형을 제공해 주는것이 아주 좋을것 같다. 그래서 언어설계가들은 모든 경우에도 받아 들일수 있는 자료형들을 만들려고 생각하였다. 이 생각은 전에도 있었고 아직까지도 있으나 그 어느 언어도 프로그람작성자들에게 사용하는데 요구되는 각이한 항목형들을 전부 제공해 주지 못하였다.

C++는 프로그람작성자들이 객체의 구체례상에서 수행될수 있는 연산범위들을 모두 포함하는 새로운 자료객체들을 선언할수 있게 해준다. 그리고 프로그람작성자는 다른 프로그람작성자들이 정의한 객체들도 물론 사용할수 있다.

5.2 객체와 통보문, 메쏘드

승용차를 하나의 객체로서 생각할수 있다. 승용차는 운전수가 볼수 없는 복잡한 세부들과 함께 그에 대한 처리들을 진행한다. 실례로 승용차의 속도를 높이기 위해 운전수는 가속변을 밟는다. 승용차는 통보문 《더 빨리 가시오.》를 받고 기관속도를 높이는 내부방법(메쏘드)을 불러 낸다.

승용차를 모는 이 설명에서 많은 객체지향개념들이 리용되였다. 이 객체지향개념들은 다음과 같다.

객체	은페된 내부구조를 가지는 항목이다. 사용자는 객체에 통보문을 보내며 이때 불러 낸 내부메쏘드에 의해 그 은페구조가 조작되거나 호출된다.
통보문	메쏘드를 불러 내기 위하여 객체에 보내지는 요구이다.
메쏘드	객체의 내부상태를 조작하거나 접근하는 동작들의 모임이다. 객체사용자는 이 동작들의 실행을 볼수 없다.

5.3 클라스

C++에서 객체란 클라스의 구체례를 말한다. 객체는 변수들과 그 변수들을 조작하는 코드를 하나의 단일한 이름으로 된 항목으로 결합시킨다.

중요한것은 클라스가 객체의 명세라는것이며 그것은 사실상 기억기에서 수행될 연산들 혹은 메쏘드들의 서술과 함께 결합된 기억기의 서술이라는것이다. 그 어떤 물리적인 연산(조작)들이 실현되자면 먼저 그 클라스의 구체례가 만들어져야 한다. 매 물리적표시에는 제공된 연산들로써 조작할수 있는 자료항목들의 사본이 들어 있다. 객체를 도식으로 표시하면 그림 5-2와 같다.

이 실례에서 자료에 대한 연산을 메쏘드라고 한다. 쏘프트웨어객체를 쓰면 프로그람형식을 훨씬 더 명백하게 할수 있고 다른 프로그람에 있는 클라스도 사용할수 있다.

주의: 코드와 자료를 하나의 클라스로 묶는 개념을 흔히 교갑화(encapsulation)라고 한다.

자 료

메쏘드 1

메쏘드 2

메쏘드 3

내부은폐자료를 호출하는
메쏘드들을 불러 내기 위해
객체에 보내지는 통보문

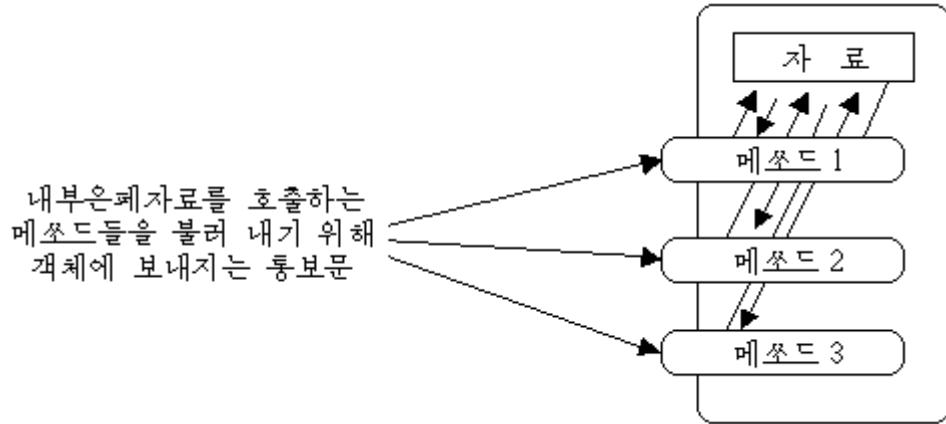



그림 5-2. 객체들에서 연산략도

5.3.1 은행구좌를 위한 객체

은행구좌를 표현하는 객체의 실현을 자세히 보기전에 이러한 객체에 보내져야 할 통보문들을 고찰하자. 아주 간단한 형식의 은행구좌에 대하여 다음과 같은 통보문들이 있을것이다.

- 구좌잔고를 제시한다.
- 구좌에 돈을 저금한다.
- 구좌에 대한 최소잔고를 설정한다(초과출금한계).
- 구좌를 설정한다.
- 구좌에서 돈을 내준다.

매개 통보문들은 다음의 책임을 가진 메쏘드들을 불러 낸다.

메쏘드/구축자	책임
Account	구좌(구축자)의 초기상태를 설정한다.
account_balance	구좌의 잔고를 돌려 준다.
deposit	구좌에 돈을 저금한다.
set_min_balance	초과출금한계를 설정한다. 0.00 : 초과출금한계가 없다. -10.00 : 10.00 파운드의 초과출금을 허락한다.
withdraw	구좌에 충분한 자금이 있거나 초과출금한계가 허락될 때 돈을 내준다.

객체의 상태는 두개의 float 변수들로 표현된다.

구체레변수	표현
the_balance	현재잔고.
the_min_balance	초과출금한계. -200.00값은 구좌에서 200.00파운드의 초과출금을 허락한다.

주의: 객체사용자는 구체레변수들을 볼수 없다.
구체레변수들에 대해서는 그 역할을 강조하기 위해 the_를 앞에 붙였다.

실레로 Account객체 mike에 100.00파운드를 저금하기 위하여 다음과 같이 서술할수 있다.

```
mike.deposit(100.00);
```

이것은 파라메터 100.00파운드를 가진 통보문 deposit를 객체 mike에 보내는것으로 해석된다. 또 다르게 해석한다면 통보문을 받는 사람으로서 객체 mike를 표시하는것이다. 이 경우에 객체 mike는 자기의 구좌에 100.00파운드를 저금한다고 말한다.

구좌에서 돈을 내주기 위하여 프로그람작성자는 내줄 돈량을 표시하는 파라메터를 가진 통보문 withdraw를 객체 mike에 보낸다. 메쏘드의 실현부는 mike가 자기구좌에 출금할 량만큼 충분한 자금이 있는가를 검사한다. 만일 자금이 불충분하거나 초과출금능력을 허락하지 않는다면 0.00파운드량을 돌려 준다. 실레로 mike의 은행구좌에서 20.00파운드를 출금하기 위해 다음과 같이 서술한다.

```
float obtained = mike.withdraw(20.00);
```

변수 obtained에는 은행구좌에서 출금된 실지량이 들어 있다. 이 명령문은 mike구좌에서 20.00파운드를 출금할수 있는가를 mike에게 물어 보는것으로 생각할수 있다. 객체 mike는 이때 그 돈을 출금하였다고 대답한다.

5.3.2 종합서술

다음의 프로그람은 Account의 구체레들인 객체 mike와 cori에 이 통보문들을 보내는것에 대하여 서술한다.

```
# include <iostream>
# include <iomanip>

// 클라스 Account 의 명세부와 실현부

int main( )
{
  Account mike, cori;
  float obtained;
```

```
  std::cout << std::setiosflags(std::ios::fixed);                    // x .y 형식
  std::cout << std::setiosflags(std::ios::showpoint);                // 0.10
  std::cout << std::setprecision(2);                                 // 소수부 2 자리

  mike.deposit(100.00);
  cori.deposit(120.00);

  std::cout << “ Mike`s account   = ” << mike.account_balance( ) << “ \n ”;
  std::cout << “ Corinna`s account = ” << cori.account_balance( ) << “ \n ”;

  mike.set_min_balance(-100.00);
  std::cout << “ Mike withdraws  : ” << mike.withdraw(120.00) << “ \n ”;
  std::cout << “ corinna withdraws: ” << cori.withdraw(20.00)  << “ \n ”;

  std::cout << “ Mike`s account   = ” << mike.account_balance( ) << “ \n ”;
  std::cout << “ Corinna`s account = ” << cori.account_balance( ) << “ \n ”;
  return 0;
}
```

주의: 이 프로그람은 클라스 Account의 명세부를 포함시켜야 콤파일된다.

이 프로그람을 콤파일하고 실행하면 결과는 다음과 같다.

```
Mike`s account      = 100.00
Corinna`s account   = 120.00
Mike withdraws      : 120.00
Corinna withdraws   : 20.00
Mike`s account      = -20.00
Corinna`s account   = 100.00
```

5.3.3 실행문제

클라스 Account는 구좌의 잔고를 보유하기 위하여 float를 사용한다. 이것은 그리 좋은 방법이 못된다. 왜냐하면 류점수는 소수부가 정해 진 수만 보유하기때문이다. 그러므로 구좌에 큰 잔고가 들어 있을 때는 둥그리기오유가 있게 된다.

5.4 함 수

클라스에서 메쏘드들은 C++함수로서 실현된다. 함수(function)는 간단히 말하여 어떤 동작을 수행하기 위해 실행될수 있는 코드모임이다. 실행시에 함수는 클라스에서 선언된 자료항목들을 호출하며 그 함수에 파라메터로서 넘겨 진 정보가 만들어 진다. 매 객체에서 정의된 변수들의 현재내용들은 객체의 상태를 반영한다. 우의 실례의 은행구좌에는 다음과 같은 함수들이 있다.

메쏘드 (함수로서 실현)	은페자료항목들 (객체상태를 표시)
Account account_balance withdraw deposit set_min_balance	the_balance the_min_balance

Account 는 특수한 경우이다. Account 는 클라스와 같은 이름을 가지는데 이 것 을 구 축 자 (constructor)라고 한다. 이 함수는 클라스의 구체례가 창조될 때 자동적으로 호출된다. Account 클라스에서 구축자는 개인은행구좌에 초기량을 ￡0.00 로, 최소잔고를 ￡0.00 로 설정하는 책임을 가진다.

다음의 선언으로 콤파일러는 구축자 Account 를 호출하는 코드를 삽입한다.

```
Account  mike;
```

구축자함수 Account 의 실현부는 다음과 같다.

```
Account :: Account( )
{
    the_balance = the_min_balance = 0.00;
}
```

주의: 구축자는 값을 돌려 주지 않는다.
구축자함수 Account 가 클라스 Account 의 성원이라는것을 보여 주기 위하여 유효범위해결연산자 ::를 사용한다.

함수 account_balance 는 아주 전형적인 C++함수인데 개인은행구좌에 현금을 넣는데 쓰인다.

```
float Account ::account_balance( )
{
  return the_balance;
}
```

함수의 돌림결과형은 함수이름앞에 쓰는데 우의 경우는 float 이다. 즉

결과형

```
float Account ::account _balance( )
{
    return the_balance;
}
```

넘겨 주는 실제값

통보문 account_balance 가 객체에 보내질 때 C++함수 account_balance 가 실행된다. 실례로 다음의 식은 객체 mike 의 잔고를 인쇄한다. 통보문 account_balance 를 객체 mike 에 보낸 결과는 구좌의 현재잔고를 표시하는 float 값이다.

```
std::cout << mike.account_balance( );
```

주의: 함수에서 제공되는 the_balance 는 객체 mike 의 the_balance 이다. 클라스 Account 의 구체례인 매 객체들은 자기에게 고유한 the_balance 의 복사품을 가질것이다.

5.4.1 함수의 파라메터

함수는 임의의 개수의 값(실제파라메터)들을 받을수 있으며 함수가 연산하는 자료항목은 변할수 있다. 아래경우에서는 구좌에서 출금될 돈을 넘기는데 한개의 값만이 사용된다. 그림 5-3 은 withdraw 함수의 배치도를 보여 준다.

C++에서 함수의 파라메터는 없을수도 있고 여러개일수도 있으나 그 함수의 결과는 오직 한개이다. 함수결과로 다중항목들을 넘기는 방법들이 있는데 이 경우에는 호출한 환경에로 값들을 넘겨 주기 위하여 파라메터기구를 사용한다. return 명령문을 만나면 즉시에 그 함수에서 벗어 난다.

mike 구좌에서 20.00 파운드를 출금하기 위하여 코드를 다음과 같이 쓸수 있다.

```
Obtained = mike.withdraw(20.00);
```

주의: C++코드작성안내서들에서는 흔히 함수에 한개의 돌림명령문만을 쓰는데 그 명령문은 대체로 마지막명령문이다.

돌림결과형　　　함수에 넘기는 값

```
float Account ::withdraw(float money)
{
  if (the_balance - money >= the_min_balance)
  {
   the_balance = the_balance - money;
   return money;
  } else {
   return 0.00;
  }
}
```

돌림값

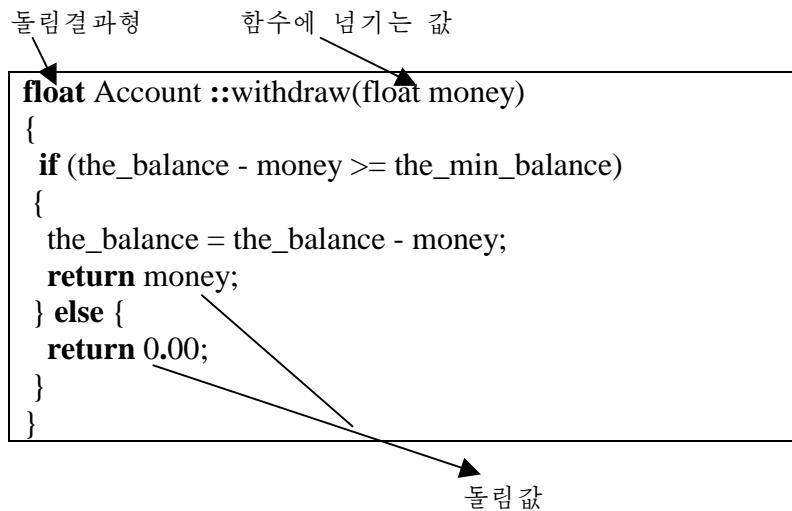



그림 5-3. 함수로서 실현되는 withdraw 메쏘드

5.4.2 void 함수

아래의 deposit 함수에서 보여 주는것처럼 함수는 값을 넘겨 주지 않아도 된다. 이 함수는 개인은행구좌에 돈을 저금한다.

```
void Account ::deposit(float money)
{
   the_balance =  the_balance + money;
}
```

주의: void 함수는 함수탈퇴명령문 return 을 쓸수 있다.

실례로 다음의 코드토막은 mike 의 은행구좌에 250 파운드를 저금한다.

```
mike.deposit(250.00);
```

5.5 클라스 Account 의 명세부와 실현부

Account 를 표시하는 C++클라스는 두개의 구성요소로 되여 있다.

- 클라스의 명세부(specification)
  클라스의 명세부는 클라스가 무엇을 하는가에 대해 서술한다. 그러나 그것이 어떻게 실현되는가는 서술하지 않는다. 프로그람작성자는 클라스가 무엇을 수행하는가에 대한 추가문서(설명문)를 달수 있다.
- 클라스의 실현부(implementation)
  클라스의 실현부는 통보문이 클라스의 구체례에 보내질 때 호출되는 메쏘드들의 실현부를 서술한다. 이것은 일반적으로 독립적으로 콤파일된 다음 사용자프로그람에 련결되기때문에 클라스를 사용하고 있는 프로그람작성자에게는 보이지 않는다.

5.5.1 클라스 Account 의 명세부

클라스 Account 의 명세부는 다음과 같다.

```
class Account{
public:
  Account( );
  float account_balance( );                    // 잔고를 돌려 준다
  float withdraw(float);                       // 구좌에서 출금
  void deposit(float);                         // 구좌에 저금
  void set_min_balance(float);                 // 최소잔고설정
private:
  float the_balance;                           // 현재잔고
  float the_min_balance;                       //  최소잔고
};
```

이것은 Account 클라스의 구체례에 보내질수 있는 통보문들의 이름을 서술한것이다. 이것들은 클라스의 메쏘드들을 실현하는데 쓰이는 C++의 함수원형(function prototype)들이다.

Account 클라스의 명세부는 두개의 명백한 부분 즉 프로그람작성자에게 보이는 공개부(public)와 프로그람작성자에게 보이지 않는 비공개부(private)로 되여 있다.

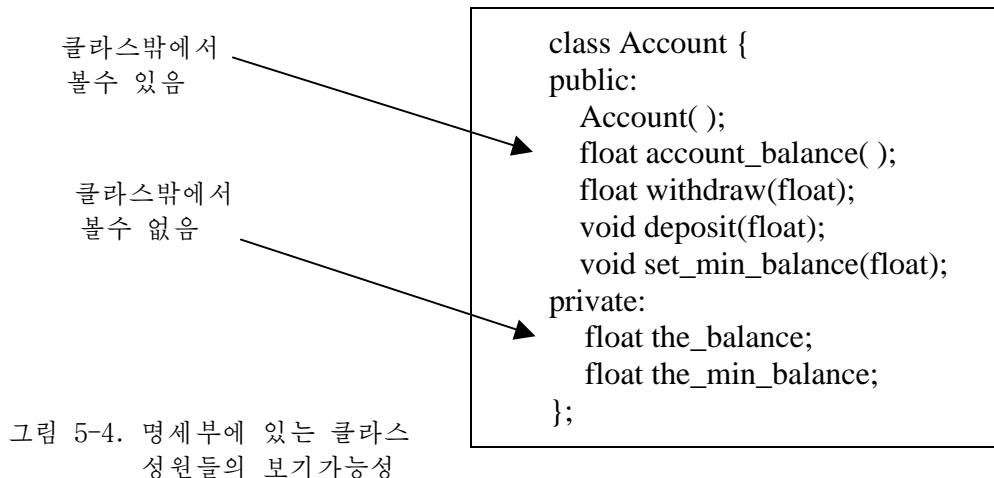
여기서 객체가 사용하는 기억장소는 클라스의 명세부에서 선언되여야 한다. 기억장소는 클라스의 비공개부에서 선언되기때문에 프로그람작성자는 호출할수 없다. 이것은 콤파일러가 Account 클라스의 매 객체를 창조하는데 얼마나 많은 기억공간을 할당하는가를 알게 하기 위해서이다. 17.13 에서 이 기억장소를 은페하는 메쏘드에 대하여 서술한다.

클라스의 성원들에 대한 호출권리는 보기가능성변경자(visibility modifier)들인 public 와 private 에 의해 조종된다. 그림 5-4 에서 클라스의 보기가능항목을 보여 준다.

클라스밖에서 볼수 있음 → public 성원

클라스밖에서 볼수 없음 → private 성원

```
class Account {
public:
   Account( );
   float account_balance( );
   float withdraw(float);
   void deposit(float);
   void set_min_balance(float);
private:
    float the_balance;
    float the_min_balance;
};
```

그림 5-4. 명세부에 있는 클라스 성원들의 보기가능성

기정값에 의해 클라스에 있는 모든 성원들은 비공개로 되여 있다가 표식 public 가 나타난 다음부터 클라스밖에서 볼수 있다.

주의: 프로그람작성에서는 일반적으로 클라스자료성원들을 비공개로 만들며 의뢰자가 자주 사용하는 클라스의 메쏘드만을 공개로 만든다.

5.5.2 클라스 Account 의 실현부

Account 클라스의 실현부에는 원형이 클라스의 명세부에 속해 있는 함수들의 본체가 들어 있다. 함수본체는 클라스 Account 에 속한다는것을 보여 주기 위하여 유효범위해결연산자 **::**를 사용한다.

```
Account ::Account( )
{
  the_balance = the_min_balance = 0.00;
}

float Account::account_balance( )
```

```
{
   return the_balance;
}
float Account ::withdraw(const float money)
{
  if (the_balance-money >= the_min_balance)
  {
     the_balance = the_balance - money;
     return  money;
  } else {
     return 0.00;
  }
}
void Account :: deposit (float money)
{
  the_balance = the_balance + money;
}
void Account :: set_min_balance(float money)
{
  the_min_balance = money;
}
```

5.5.3 Account 클라스의 전체적모습

다음의 표는 Account 클라스의 명세부와 실현부를 나란히 보여 준다.

<table>
<tr><th>명세부</th><th>실현부</th></tr>
<tr><td><pre>class Account{
public:
  Account( );
  float account_balance( );
  float withdraw(float);
  void deposit(float);
  void set_min_balance(float);
private:
  float the_balance;
  float the_min_balance;
};</pre></td><td><pre>Account ::Account( )
{
 the_balance = the_min_balance = 0.00;
}
float Account::account_balance( )
{
 return the_balance;
}
float Account:;withdraw(float money)
{
 if(the_balance-money >= the_min_balance)
  {
    the_balance = the_balance - money;
    return money;
  }else {</pre></td></tr>
</table>

```
  return 0.00;
 }
}
void Account ::deposit(float money)
{
  the_balance = the_balance + money;
}
void Account::
    set_min_balance(float money)
{
  the_min_balance = money;
}
```

콤파일러와 사용자의 견지에서 본 이 두 부분의 역할은 다음과 같다.

클라스	역할
명세부	클라스가 무엇을 하는가에 대한 세부.
실현부	클라스가 이 연산들을 어떻게 집행하는가에 대한 세부.

5.5.4 클라스구성요소의 계층

Account 클라스의 보기가능성계층을 그림 5-5 에서 보여 준다.

보기가능성

구성요소들이 들어 있는 곳

의뢰자가 클라스를 볼수 있음

deposit
withdraw
acconut balance
set_min_balance

클라스명세부의 공개부

의뢰자가 클라스를 볼수 없음

the_balance
the_min_balance

클라스명세부의 비공개부

함수들의 실현부

클라스의 실현부

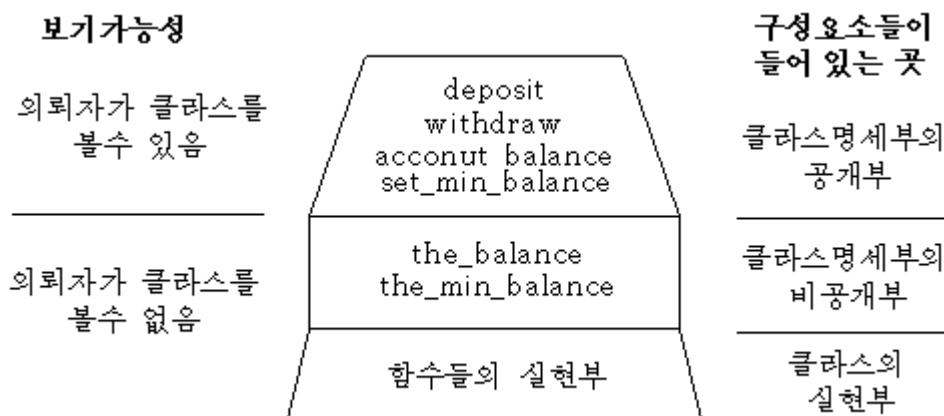



그림 5-5. Account 클라스의 메쏘드들과 구체례속성들의 보기가능성

5.5.5 용어

클라스의 구성요소들을 서술하는데 다음의 용어들이 리용된다.

용어	Account 클라스의 실례	설명
자료성원 (구체례속성△)	the_balance	객체의 상태를 나타낸다. 자료성원에 대한 선언은 명세부의 비공개부에 있다. 이 책에서는 자료성원의 이름앞에 the_를 붙인다.
메쏘드 (구체례메쏘드△ 메쏘드△)	Deposit	객체의 자료성원들을 호출하기 위해 사용되는 함수이다.

주의: △이 붙은 용어는 Smalltalk 언어에서 나왔다.

5.6 개인구좌관리프로그람

LCD 현시장치를 가진 작은 노트형콤퓨터에서 실행되는 개인구좌관리프로그람은 영상표시장치에 안내서를 표시한다. 즉

```
B  - Balance
W  -Withdraw
D  -  Deposit
E  -  Exit
Input selection:
```

마감행은 사용자가 요구되는 입력값들을 넣을수 있게 하며 그 결과들을 표시하는 칸을 준다.

실례로 그 화면은 아래와 같으며 이때 사용자입력은 굵은 글자로 보여 준다. 실례로 D 항목을 선택하여 사용자가 100 파운드를 입력하면 화면에 다음과 같은 결과가 나온다.

Amount to deposit = **100**
Balance = 100

사용자가 W 항목을 선택하고 80 파운드를 출금하였다고 지정하면 아래부분에 다음과 같은 결과가 나온다.

Amount to withdraw = **80**
Balance = 20

마지막으로 사용자는 B 항목을 선택하여 현재잔고를 볼수 있다. 이 경우에 다음과 같은 결과가 나온다.

```
Balance = 20
```

5.6.1 프로그램

Account 클라스의 명세부와 실현부를 제외한 프로그람의 완성부분은 다음과 같다. 일반적으로 모의은행구좌에 보관된 량을 분실할수 있으므로 프로그람은 절대로 끝을 맺지 않는다.

```
# include <iostream>
# include <iomanip>

// 클라스 Account 의 명세부와 실현부

int main( )
{
std::cout << std :: setiosflags (std::ios::fixed);          // x . y 형식
std::cout << std :: setiosflags (std::ios::showpoint);      // 0.10
std::cout << std :: setprecision (2);                       // 소수부 2 자리

Account mine;
```

프로그람은 수행되여야 할 업무들의 안내서를 표시하기 위하여 반복적으로 순환한다.

```
    while ( true )
    {
        char action;
        float obtain, process ;

        std::cout << “ \n” ;
        std::cout << “ B - Balance”     << “ \n”  << “ \n” ;
        std::cout << “ W - withdraw ”  << “ \n” << “ \n” ;
        std::cout << “ D - Deposit ”    << “ \n” << “ \n” ;
        std::cout << “ E - Exit ”        << “ \n” << “ \n” ;

        std::cout << “Input selection:” ;
        std::cin >> action;
```

switch 명령문은 사용자가 요청할수 있는 각이한 업무들을 선택하는데 쓰인다.

```
     switch (action)
      {
          case ' B ' : case ' b ':
             std::cout << “Balance =” << mine.account_balance( ) << “ \n ” ;
             break;
```

```
                case ' W ' : case  ' w ';
                    std::cout << "Amount to withdraw : " ;
                    std::cin >> process;
                    obtain = mine.withdraw(process);
                     if  (obtain > 0.00)
                         std::cout << "Withdrawn " << obtain<< " \n" ;
                     else
                         std::cout << "Not enough funds" << " \n" ;
                     break;
                case ' D ': case ' d ':
                     std::cout << "Amount to deposit : " ;
                     std::cin >> process;
                     mine.deposit(process);
                     break;
                case ' E ' : case ' e ':
                     return 0;
                default:
                    std::cout << "Invalid selection" << " \n" ;
            }
        }
        return 0;
}
```

주의: 입력자료는 검사되지 않으며 타당한 수이든 아무런 수이든 관계 없다. 실례로 구좌에서 부수값량을 출금할수도 있다.

5.7 검토자와 변이자

클라스의 메쏘드는 검토자(inspector)일수도 있고 변이자(mutator)일수도 있다. 이 메쏘드들의 역할은 아래표와 같다.

메쏘드	메쏘드의 역할	Account 클라스의 실례
검토자	객체상태를 변화시키지 않는다.	account_balance
변이자	객체상태를 변화시킨다.	Withdraw deposit

5.7.1 메쏘드를 검토자로 만들기

성원함수가 검토자라는 약속은 그 메쏘드뒤에 예약어 const 를 붙여 실현할수 있다. 실례로 account_balance 메쏘드가 검토자라는것을 약속하기 위해 선언을 다음과 같이 한다.

```
float account_balance( ) const;                    // 잔고를 돌려 준다
```

그리고 이 선언의 실현부는 다음과 같다.

```
float Account::account_balance( ) const
{
    return the_balance;
}
```

콤파일러는 메쏘드가 클라스의 어떤 자료성원도 변화시키지 않는다는것을 검사한다. const 메쏘드에서 자료성원을 변화시키는 시도가 있게 되면 콤파일할 때 오유통보문이 발생된다.

5.8 명세부와 실현부를 함께 서술하기

클라스의 명세부와 실현부는 자체로 결합될수 있다. 본질상 메쏘드의 본체는 클라스의 명세부에 포함되여 있다. 실례로 Account 클라스는 다음과 같이 쓸수 있다.

```
class Account {
public:
  Account ( )
  {
      the_balance = the_min_balance = 0.00;
  }

  float account_balance ( ) const
  {
      return the_balance;
  }

  float withdraw (const float money)
  {
       if (the_balance – money >= the_min_balance)
       {
         the_balance = the_balance - money;
         return money;
       } else {
        return 0.00;
       }
  }

  void deposit(const float money)
  {
    the_balance = the_balance + money;
  }
```

```
  void set_min_balance( const float money )
  {
    the_min_balance = money;
  }
private:
  float the_balance;                          // 구좌의 잔고
  float the_min_balance;                      // 최소잔고(초과출금한계)
};
```

그러나 이렇게 하면 클라스의 사용자에게 실현부코드가 보여 지므로 코드가 조잡해 보인다.

5.9 자체평가

- Account 클라스의 구체례를 어떻게 선언하는가?
- 클라스의 구체례가 선언될 때 어떤 코드가 실행되는가?
- 프로그람작성자는 클라스구체례에 통보문을 보내는 코드를 어떻게 만드는가?
- 클라스에 무엇이 포함되여 있는가?
- 클라스의 명세부와 실현부를 갈라서 서술하는 우점은 무엇인가?
- 왜 클라스에 그 클라스와 같은 이름을 가진 메쏘드가 있어야 하는가?
- 자료와 그 자료를 조작하는 코드가 함께 들어 있는 우점은 무엇인가?
- 클라스에 들어 있는 메쏘드를 비공개부로 할수 있는가? 왜 그런가?
- 클라스에 들어 있는 메쏘드를 공개부로 할수 있는가? 왜 그런가?
- 클라스에서 비공개부자료성원들을 호출하기 위하여 어떻게 배렬할수 있는가?
- 클라스안에 클라스를 선언하고 싶을 때가 있겠는가?

5.10 련 습

다음의 클라스들을 작성하시오.

- Safer_Account
  앞에서 본 은행구좌클라스는 넘겨 받는 파라메터값검사에서 약간한 오유가 있었다. 오유검사를 정확하게 하는 클라스로 다시 만드시오. 실례로 은행구좌에 부수값의 돈은 저금할수 없도록 작성하시오.
- Perfomance
  극장에서 개별적인 공연에 대한 좌석상태를 서술하는 클라스를 작성하시오.

이 클라스는 다음의 메쏘드를 가진다.

메쏘드	책임
available	공연을 위해 가능한 n 개의 좌석들이 있다면 true 를 돌려 주고 그렇지 않으면 false 를 돌려 준다.
book	n 개의 좌석들을 예약한다. 예약된 좌석수를 돌려 준다. 이때 0 결과는 실패를 의미한다.
cancel	이미 예약된 좌석들에 대한 예약취소.
remaining	이 공연을 위해 예약되지 않은 좌석수를 돌려 준다.

다음의 프로그람을 구성하시오.

- 검사프로그람
  Safer_Account 클라스를 검사하기 위한 프로그람.
  Performance 클라스를 검사하기 위한 프로그람.
- 극장
  그날 극장예약을 그날로 진행하는 경영업무처리프로그람.
  공연은 매일 3 번씩 한다. 1 시에 오후공연, 5 시에 저녁공연, 8 시 30 분에 기본공연을 한다. 프로그람은 이 세개의 공연들중 임의의 공연에 대해 극장좌석예약을 조작할수 있고 특별공연을 위해 좌석을 남겨 두는것과 같은 세부내용들을 줄수 있다.

6 함 수

이 장에서는 C++프로그람의 함수에 대해 서술한다. 일반적으로 함수는 클라스의 성원들이다. 그러나 함수는 교갑화된 구조의 밖에서 효과 있게 사용될수도 있다.

6.1 소 개

앞에서 서술한것처럼 함수(function)는 간단히 말하여 프로그람의 어떤 해당한 부분에서 호출될수 있는 C++식들을 함께 묶은것이다. 프로그람작성자는 문제풀이를 추상화할수 있다. 함수들은 클라스의 자료항목들과 결합될 때 더 위력하게 쓰일수 있다.

함수에 대해서는 겹쳐쓰기(nest)를 할수 없다. 다시 말하여 함수안에 다른 함수가 선언될수 없다. 왜냐하면 그래야 프로그람의 실행을 위한 실현을 간단하게 할수 있고 자료항목들에 대한 호출을 아주 효률적으로 할수 있기때문이다.

파라메터로 넘겨 받는 두 류점수값을 더하는 함수를 그림 6-1에서 보여 준다.

함수에 넘기는 값

결과형

국부변수

```
float sum ( float first, float second )
{
    float answer;
    answer = num1 + num2;
    return answer;
}
```

돌림값 answer

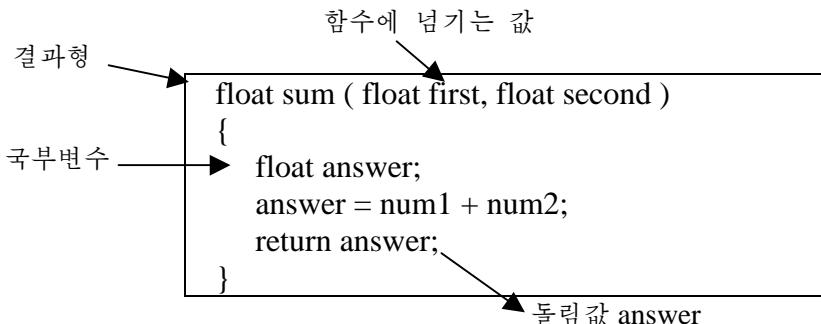



그림 6-1. 두 파라메터의 합을 돌려 주는 함수

6.1.1 함수사용법

다음의 프로그람은 사과와 귤의 무게를 합하여 표시한다.

```
#include  <iostream>
float sum ( float first, float second)
{
        float answer;
        answer = first + sescond;
        return answer;
}

int main ( )
{
```

```
        float apples_weight   = 1.21;                                  // kg으로
        float oranges_weight = 1.51;

        std::cout << "Total weight of fruit in kilograms is "
                  << sum( apples_weight, oranges_weight ) << "\n";
        return 0;
}
```

주의: 넘겨 지는 값은 직접값(literal value)일수도 있다. 실례로
std :: cout << sum(10.2, 3.4) << "\n";

이 프로그람을 콤파일하고 실행하면 결과는 다음과 같다.

```
Total weight of fruit in kilograms is 2.72
```

6.1.2 용어

다음의 용어는 함수호출의 처리를 서술할 때 사용된다.

용어	설 명	실 례
실제파라메터	함수에 넘겨 지는 변수 혹은 직접값	apples_weight
형식파라메터	함수내부에서 호출될 때의 파라메터이름	first

sum 함수가 호출될 때 2개의 실제파라메터들인 apples_weight와 oranges_weight가 넘겨 진다. 함수의 실현부에서 형식파라메터들인 first와 second는 그에 대한 호출이 요구될 때 사용된다.

6.1.3 함수에서 국부변수

변수가 함수내부에서 선언될 때 그의 수명은 함수의 수명과 같다. 함수가 호출될 때 변수기억령역은 실시간탄창에 자동적으로 만들어 진다. 함수가 탈퇴될 때 그 변수기억령역은 체계에 다시 넘겨 진다.

주의: 프로그람작성에서는 일반적으로 함수가 다음의 호출만 하게 한다.

- 함수에 대한 파라메터들.
- 국부변수들.
- 이 함수가 성원으로 되는 클라스에 포함된 자료항목들.

6.1.4 함수원형

C++에서 함수들은 사용되기전에 지정되여 있어야 한다. 만일 함수가 미리 정의되여 있지 않다면 함수원형(function prototype)을 리용하여 콤파일러에 그 함수의 명세(specification) 또는 서명(signature)을 알려 준다. 실례로 sum함수에 대한 원형은 다음과 같다.

```
float sum ( float , float );
```

주의: 함수가 클라스의 성원이라면 그의 명세(서명)는 클라스명세부에 들어 있다. 함수가 사용되기전에 정의되여 있다면 함수원형을 지정하지 않아도 된다. 형식파라메터의 이름은 함수원형에서 빠질수 있다.

6.1.5 void함수

함수가 결과를 돌리지 않는 경우에는 그 함수를 void함수로 선언한다. 실례로 display_name함수의 선언은 다음과 같다.

```
void display_name( );

int main ( )
{
      display_name ( );
      return 0;
}
void display_name ( )
{
       cout  <<  " A N Other  ";
}
```

주의: 함수에서 파라메터가 없는 빈 괄호들은 파라메터가 없다는것을 더 명확히 알려주는 void로 교체할수 있다. 그러나 콤파일러는 항상 호출에서 해당한 개수의 파라메터들이 정확히 사용되였는가를 검열한다. 21.13에서 파라메터의 개수가 고정되지 않은 함수를 어떻게 호출하는가에 대하여 서술한다.

6.2 값에 의한 호출과 참조에 의한 호출

C++에서 실제파라메터들은 두가지 처리를 리용하여 함수에 넘겨 지는데 그 처리들은 다음과 같다.

- 값에 의한 호출
  복사는 실제파라메터를 통하여 이루어 지며 이 복사는 함수내부에서 형식파라메터로서 사용된다. 이 방법을 사용하면 정보는 함수에 들어 갈수만 있다.
- 참조에 의한 호출
  실제파라메터의 참조는 함수내부에서 형식파라메터로서 사용된다. 형식파라메터에 대한 변화는 결과적으로 함수에 넘겨 지는 실제파라메터들을 변화시킬것이다. 이 방법을 사용하면 정보가 함수에 들어 갈수도 있고 함수에서 나올수도 있다. 이 방법은 함수에 실제파라메터의 기억주소를 넘겨 주는것에 의해 실행된다.

실례로 두 변수의 내용을 서로 바꾸는 swap함수를 쓰면 자료는 함수에 들어 갈수도 있고 나올수도 있다. 즉 아래에 있는 swap함수는 pcs_room1과 pcs_room2의 내용을 서로 바꿀것이다.

참조에 의한 호출
 	
실제파라메터들의 내용을 바꾼다.
	int pcs_room_1, pcs_room_2;
	
	void swap (int& first, int& second)
	{
	int tmp = first;
	first = second; second = tmp;
	}

값에 의한 호출
 	
실제파라메터들의 내용을 바꾸지 않는다.
	int pcs_room_1, pcs_room_2;
	
	void swap_wrong (int first, int second)
	{
	int tmp = first;
	first = second; second = tmp;
	}

주의: 참조에 의한 호출은 형식파라메터의 형뒤에 &를 붙여 서술한다.
swap함수인 경우에만 pcs_room_1과 pcs_room_2변수에 포함된 값들이 변화된다.

6.2.1 종합서술

swap_wrong함수를 사용하면 실제파라메터의 내용이 서로 교환되는 결과를 얻지 못할것이다. 그것은 실제파라메터의 복사가 함수내부에서 형식파라메터로서 사용되기때문이다. 이 형식파라메터들에 대하여 진행된 변화는 함수가 끝나면 없어 진다.

```
# include  <iostream>

void swap_wrong(int first, int second)
{
  int tmp = first;
  first = second;  second = tmp;
}

int main ( )
{
  int pcs_room_1 = 4;
  int pcs_room_2 = 8;

  swap_wrong(pcs_room_1, pcs_room_2);

  std::cout<< “PC's room 1 =  “<< pcs_room_1 <<  “\n ”;
  std::cout<< “PC's room 2 =  “<< pcs_room_2 << “\n ”;
  return 0;
}
```

이 프로그람을 콤파일하고 실행하면 결과는 다음과 같다.

```
PC's room 1 = 4
PC's room 2 = 8
```

주의: 이 프로그람에서 매방에 있는 PC들의 개수는 서로 교환되지 않는다. 왜냐하면 실제파라메터가 값에 의해 넘어 가기때문이다.

그러나 swap함수를 사용하면 실제파라메터들의 내용을 서로 교체하려는 결과를 얻을것이다.

```
# include  <iostream>

void swap ( int&  first,  int&  second )
{
  int tmp = first;
  first = second;  second = tmp;
}

int main( )
{
  int pcs_room_1 = 4;
  int pcs_room_2 = 8;

  swap( pcs_room_1, pcs_room_2 );

  std::cout <<  "PC's room 1 =  " << pcs_room_1 <<  " \n "
  std::cout <<  "PC's room 2 =  " << pcs_room_2 <<  " \n "
  return 0;
}
```

이 프로그람을 콤파일하고 실행하면 결과는 다음과 같다.

```
PC's room 1 = 8
PC's room 2 = 4
```

주의: 파라메터를 참조로 넘기는 방법은 파라메터형에 &를 붙이는것이다(실례로 int&). &에 대하여서는 17장에서 설명한다.

실제파라메터가 함수에 넘어 가는 경우 그것은 반드시 비상수값이여야 한다. 그러므로 직접값(literal value)은 참조에 의해 넘길수 없다.

6.3 함수에서 const파라메터

함수에 대하여 추가적인 안전을 담보하기 위하여 파라메터서술에 함수의 본체안에서 읽기만 할수 있다는것을 나타내는 지정자로서 const를 쓸수 있다. 프로그람작성자가 부주의로 이 형식파라메터에 대한 쓰기를 코드화하면 오유통보문이 발생된다.

```
void wrong ( const int  item, const char&  ch)
{
    item = 123;                                    // 콤파일시 오유발생
    ch = 'A';                                      // 역시 콤파일시 오유발생
}
```

주의: 참조에 의한 호출에서는 항목을 복사하지 않으며 대신 항목의 참조(기계단어에 들어 있는)가 넘어 간다.

앞붙이 const를 사용하면 함수의 코드화를 더 안전하게 할수 있다. 특히 큰 항목이 충분한 리유로 함수에 참조로 넘어 갈 때 const수식자는 실제파라메터에 대한 우연적인 변화를 막아 준다.

6.3.1 파라메터넘기기에 대한 요약

선언된 파라메터: (실례로 int형의 형식 파라메터를 사용)	항목이 무엇에 의해 넘겨 지는가	형식파라메터를 변화시킬수 있는가	돌려 줄 때 변화
int item	값	예	아니
const int item	값	아니	아니
int& item	참조	예	예
const int& item	참조	아니	아니

주의: 이것은 Ada의 IN, OUT파라메터와 비슷하다.

배렬파라메터에서의 const사용에 대해서는 17.4.1의 지적자와 배렬에 대한 파라메터호출권부분에서 서술한다.

6.4 재 귀

이전의 어떤 콤퓨터사전에는 재귀라는데 대해 다음과 같이 재미 있게 정의되여 있었다.

《재귀를 리해하지 못한 사람은 그 재귀의 첫 시작부분을 보시오.》

주의: 재귀(recursion)란 간단히 문제의 부분을 푼 다음 나머지부분을 계속 풀기 위하여 그 풀이방법을 다시 호출하는것이다. 이러한 《재귀》는 문제의 모든 부분이 풀려 질 때까지 계속된다.

6.4.1 재귀함수

재귀는 그 본체안에서 자기자체를 호출하는 구조로 되여 있는 수속이나 함수의 능력이다. 이것은 처음에는 좀 이상스럽게 보이는 개념이지만 그것을 리용하면 코드화가 간단해 질수 있고 그렇지 못하면 조잡스러워 진다.

실례로 다음과 같은 수를 한 글자씩 쓰기 위한 재귀함수를 작성하자.

수를 쓰기:(write_number)

- 수를 두 부분으로 가르기.
  (a) 첫 수자(수를 10으로 나눌 때의 나머지).
  (b) 다른 수자들(수를 10으로 나눌 때의 상).
  실례로 123은 다음과 같이 갈라 진다.
  3 (첫 수자)
  12 (다른 수자들)
- 만일 다른 수자들이 10보다 크거나 같다면 수쓰기코드를 재귀적으로 호출하여 또 다른 수자들을 쓴다.
- 첫 수자를 한개 글자로 출력한다.

호출순서는 다음과 같다.

호 출	실 현
write_number (123)	write_number (12); 첫 수자 3이 출력
write_number (12)	write_number (1); 첫 수자 2가 출력
write_number (1)	첫 수자 1이 출력

이 처리를 그림으로 표시하면 그림 6-2과 같다.

초기호출
123
가르기
12에 대한 재귀호출
12
3
가르기
1에 대한 재귀호출
1
2
가르기
그러나 요구되는 재귀호출이 없다.
1
되돌리기

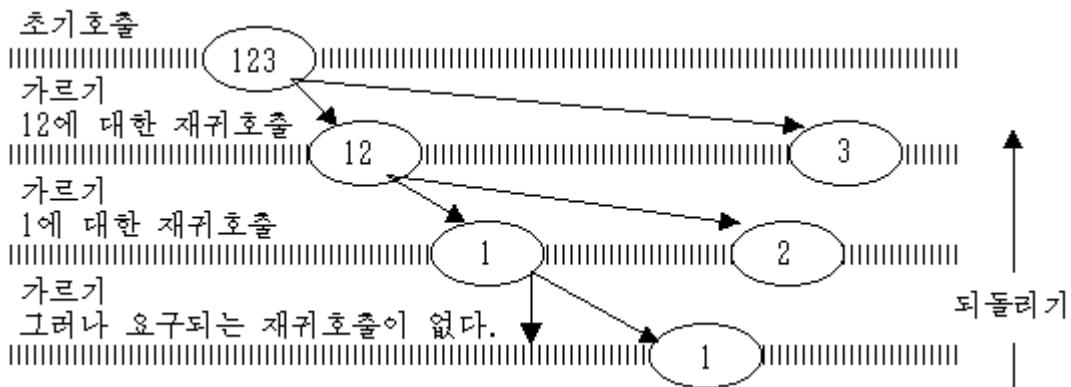



그림 6-2. 수자 123을 인쇄하는 재귀호출

이 처리는 문제의 작은 부분을 푼(이 실례에서는 한개 수자를 출력) 다음 그 코드를 재실행시켜 그 문제의 나머지부분을 푸는(이 실례에서는 다른 수자들을 출력) 방법에 기초한다. 이 실례에서는 문제의 나머지부분을 풀기전에 재귀호출이 만들어 진다. 그 나머지부분은 여전히 계속 풀려야 할 문제이며 재귀호출을 실행할 때마다 그 나머지부분(출력해야 할 수)의 규모는 점점 작아 진다.

여하튼 재귀를 동작시키기 위하여 코드는 재귀적으로 자기자체를 호출하면서 앞에서 풀어 진 문제들을 제거해야 한다. 만일 제거를 하지 않는다면 재귀호출이 계속 일어 나 체계가 재귀를 지원하기 위해 더이상 기억기를 할당할수 없으며 결과적으로 프로그람이 실패된다. 탄창공간은 재귀때마다 함수지원정보와 함께 어떤 파라메터나 국부변수를 기억하기 위해 사용된다.

```
void write_number( const int number )
{
 int number_to_print = number;
 if ( number < 0 )                                   // 부수인 경우 정수로 만든다
 {
  number_to_print= -number;  cout << “-”;
 }
 int first_digit   = number_to_print%10;             // 가르기
 int other_digits = number_to_print/10;
 if ( number_to_print >= 10 )                        // 수가 2 자리이상이면
 {
  write_number ( other_digits );                     // 다른 수자
 }
 std::cout << (char)  (first_digit + (int) ‘0’);     // 첫 수자
}
```

6.4.2 종합서술

함수 write_number 는 프로그람에서 다음과 같이 사용될수 있다.

```
int main ( )
{
 write_number (12345);
 cout << “ ”;
 write_number(-123);
 cout << “\n ”;
 return 0;
}
```

이 프로그람을 콤파일하고 실행하면 결과는 다음과 같다.

```
12345  -123
```

6.5 내부전개코드와 외부전개코드

콤파일러는 C++프로그람을 위한 실행가능코드를 구성할 때 그 함수의 코드를 생성하는데서 두가지 방법을 쓴다. 이 방법에는 내부전개(inline)와 외부전개(out of line)가 있다. swap 함수를 호출하기 위한 코드를 생성할 때의 효과들의 차이를 보면서 이 두가지 방법의 원리를 아래에서 설명한다.

- inline
  swap 함수에 대한 코드본체의 복사는 함수를 호출할 때마다 생성된다. 이것은 그림 6-3과 같다.

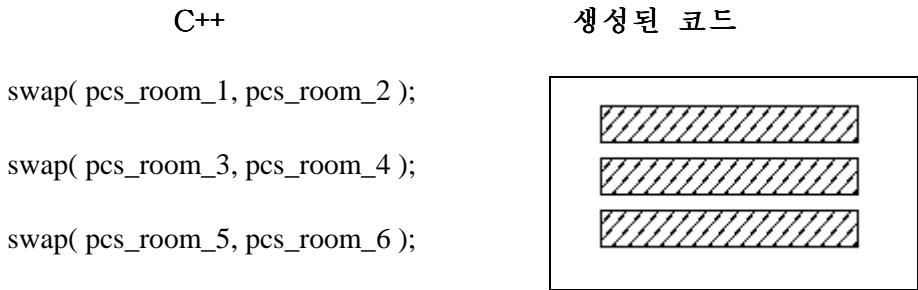
C++	생성된 코드
swap(pcs_room_1, pcs_room_2);	
swap(pcs_room_3, pcs_room_4);	
swap(pcs_room_5, pcs_room_6);	

swap를 실현하기 위한 코드

그림 6-3. 내부전개기능으로 생성된 코드

- out of line
  swap 함수에 대한 코드본체의 복사는 한번 생성되며 이 단일복사에 대한 호출들은 함수가 호출되는 위치에 놓인다. 이것은 그림 6-4와 같다.

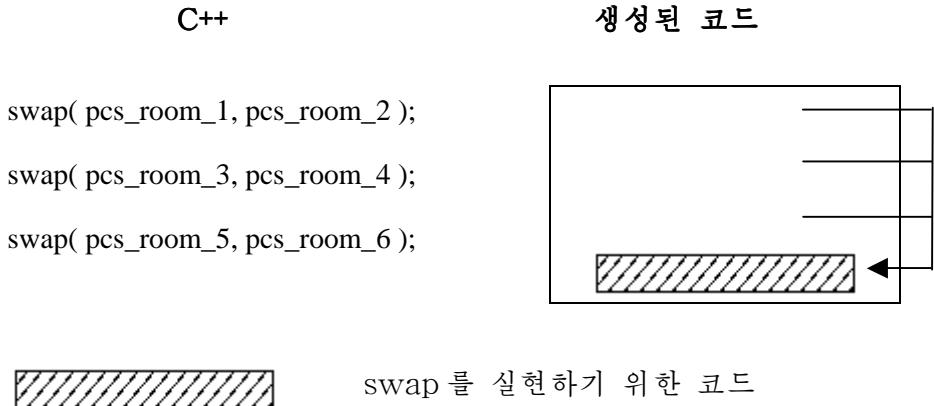
C++	생성된 코드
swap(pcs_room_1, pcs_room_2);	
swap(pcs_room_3, pcs_room_4);	
swap(pcs_room_5, pcs_room_6);	

swap를 실현하기 위한 코드

그림 6-4. 외부전개기능으로 생성된 코드

6.5.1 코드생성방법이 내부전개이거나 외부전개일 때

함수에 대한 실현방법의 선택은 그 함수가 어디에서 어떻게 선언되였는가에 달려 있다. 프로그람작성자는 직접 콤파일러가 내부전개코드를 콤파일하도록 지령을 줄수도 있다. 그러나 콤파일러는 이 권고를 거절한다.

일반적으로 내부전개 혹은 외부전개로 선언하기 위하여 다음과 같이 한다.

내부전개형식의 코드

- 함수선언앞에 예약어 inline 을 붙인다. 함수본체의 선언은 그 함수를 사용하기전에 먼저 진행된다.
- 메쏘드의 본체는 클라스명세부안에서 선언된다. 이 선언형식의 실례에 대해서는 5.8 을 보시오.

외부전개형식의 코드

- 메쏘드의 본체는 클라스명세부의 밖에서 선언된다.
- 일반함수선언.

함수가 매우 크거나 순환구조를 가지고 있거나 재귀적인 경우에는 콤파일러는 일반적으로 inline 지령을 거절한다.

6.5.2 우점과 결함

다음의 표는 함수코드를 내부전개 혹은 외부전개방식으로 두는것이 가지는 우결함에 대하여 보여 준다.

	우점	결함
내부전개코드	빨리 실행된다.	잠재적으로 코드규모가 커진다.
외부전개코드	코드규모가 작다.	보다 천천히 실행된다.

6.5.3 내부전개기능의 실례

아래에서 보여 준 프로그람에서는 swap 함수를 호출할 때마다 그 함수의 본체가 호출된다.

```
# include <iostream>

inline void swap (int &first, int &second)
{
  int tmp = first;
  first = second; second = tmp;
}

int main( )
{
  // swap 를 호출
}
```

주의: 함수본체는 함수에 대한 첫 호출이 있기전에 먼저 정의되여야 한다.

6.6 함수의 다중정의

다중정의(overloading)는 프로그람에서 여러개의 처리들을 같은 이름으로 표시하게 한다. 콤파일러는 제공된 내용이나 다른 추가정보를 사용하여 해당한 처리를 선택한다. 함수이름이 다중정의된 실례는 다음과 같다.

```
# include <iostream>
# include <string>
```

```
void what_is_this( const char);
void what_is_this( const int);
void what_is_this( const std::string);
void what_is_this( const float);
void what_is_this( );
```

주의: string 클라스의 구체례를 서술하는 표준클라스 std::string 을 사용하면 문자렬비교와 값주기에서 세련된 문자렬조작을 할수 있다. 부록 4 에 이 클라스의 성원들을 보여 준다.

```
void  what_is_this ( const char c)
{
    cout <<  "Is a character value  " << c << "\n" ;
}

void what_is_this ( const int i)
{
    cout <<  "Is an integer value  "  << i << "\n" ;
}

void what_is_this( const float f)
{
   cout <<  "Is a float value "  << f << "\n" ;
}

void what_is_this( const std::string str)
{
    cout <<  "Is a string value  "  << str << "\n" ;
}

void what_is_this( )
{
    cout <<  "A function with no parameters  " << "\n" ;
}
```

```
int main ( )
{
    what_is_this (1);
    what_is_this( (float) 1.1);                    // 1.1 은 double 상수이다
    what_is_this( ‘A’ );
    what_is_this( “Hello ” );
    cout << “\n” ;
    return 0;
}
```

실행하면 what_is_this 에 넘겨 지는 인수의 형과 값이 인쇄된다.

```
Is an integer value 1
Is a float value 1.1
Is a character value A
Is a string value Hello
```

물론 이렇게 하자면 실제로 호출되는 함수는 매 경우마다 서로 달라야 한다. 이름 what_is_this 는 5 개의 각이한 함수들로 다중정의된다. 콤파일러는 함수의 파라메터형과 개수를 보고 어느 함수를 호출할것인가를 결정한다.

6.7 한 함수에 각이한 개수의 파라메터들을 넘겨주기

콤파일러가 다중정의된 이름들을 구별할수 있는것으로 하여 아래의 실례에서 larger 함수는 더 많은 개수의 파라메터들을 받을수 있도록 작성될수 있다.

```
# include < iostream>

int larger ( const int, const int );
int larger ( const int, const int, const int );

int larger ( const int a, const int b)
{
  return a > b ? a : b;
}

int larger( const int a, const int b, const int c)
{
  return larger(a, b) < c ? larger(a, b) : c;
}
```

주의: 두개의 파라메터를 가진 함수를 더 크게 만들어 3 개의 파라메터를 가진 larger 함수를 정의한다. 콤파일러는 int 형실제파라메터의 개수에 의해 어느 larger 함수를 호출하겠는가를 결정한다.

6.7.1 종합서술

```
int main ( )
{
   int  num1 = 5,  num2 = 4,  num3 = 6;
  cout  <<  " Of  " <<  num1 <<  " , " << num2;
  cout  <<  " the  larger  is " << larger( num1,  num2 )  <<  "\n" ;
  cout  <<  " Of  " <<  num1 <<  " , " <<  num2  <<  " , " <<  num3 ;
  cout  <<  " the  larger  is " << larger( num1,  num2,  num3 )  <<  "\n" ;
  cout  <<  "\n" ;
  return  0;
}
```

이 코드의 실행결과는 다음과 같다.

```
Of  5, 4  the  larger  is  5
Of  5, 4, 6  the  larger  is  6
```

6.8 파라메터에 대한 기정값

만일 형식파라메터에 기정값을 주면 함수를 호출할 때 실제파라메터를 생략할수 있다. 다만 형식파라메터가 기정값이면 그뒤에 오는 형식파라메터들이 모두 기정값을 가져야 한다는것이다.

함수파라메터들의 합을 돌려 주는 sum 함수는 다음과 같다.

```
int sum (  const int, const int = 0, const int = 0);                // 명세부

int sum (  const int a,  const int b, const int  c)                // 실현부
{
  return  a + b + c;
}
```

주의: 여기서는 sum 함수의 두번째, 세번째 파라메터에 기정값을 주었다.
sum 함수실현부에서 기정값들을 반복해서 쓰면 오유가 생긴다.

6.8.1 종합서술

```
int main ( )
{
  int num1 = 5,  num2 = 4,  num3 = 6;
  count  <<  " Of  " << num1;
  count  <<  " the sum is  " << sum( num1 ) <<  "\n  " ;
```

```
count << " Of " << num1 << " , " << num2 ;
count << " the sum is " << sum( num1, num2 ) << "\n " ;
count << " Of " << num1 << " , " << num2 << " , " << num3 ;
count << " the sum is " << sum( num1, num2, num3 ) << "\n " ;
count << "\n" ;
return 0;
}
```

실행하면 결과는 다음과 같다.

```
Of 5 the sum is 5
Of 5, 4 the sum is 9
Of 5, 4, 6 the sum is 15
```

sum 함수를 콤파일하여 생성된 실제코드에는 세개의 실제파라메터들의 넘기기가 들어 있다.

프로그람작성자가 쓴 함수	콤파일러가 발생시킨 함수
sum(num1)	sum(num1, 0, 0)
sum(num1, num2)	sum(num1, num2, 0)

6.9 함수선언과 함수호출의 정합

실제파라메터들을 호출된 함수의 선언형으로 변환하기 위하여 4.14 에서 서술한 형변환모형을 사용한다.

두 함수가 다음과 같이 선언되였다.

```
void print_this_as_int( const int );
void print_this_as_double( const double );

void print_this_as_int ( const int i )
{
 std::cout << "is an integer value " << i << "\n" ;
}

void print_this_as_double (const double i )
{
 std::cout << "Is an double value " << i << "\n" ;
}
```

만일 다음과 같이 서로 다른 형들로 호출된다면

```
int main ( )
{
  print_this_as_int (1);
  print_this_as_int ( (short) 2) ;
  print_this_as_int ( (unsigned short) 3) ;
  print_this_as_int ( ‘A’ ) ;

  std::cout << “\n” ;
  print_this_as_double ((short) 1);
  print_this_as_double (2);
  print_this_as_double ( (float) 2.2) ;
  print_this_as_double ( (double)3.3) ;
  return 0;
}
```

결과는 다음과 같다.

```
Is an integer value 1
Is an integer value 2
Is an integer value 3
Is an integer value 65

Is an double value 1
Is an double value 2
Is an double value 2.2
Is an double value 3.3
```

주의: 문자상수 ‘A’는 int 로 형변환된다.

알림

C++는 다음과 같은것도 허락한다.

```
int main ( )
{
   print_this_as_int( (float) 1.1);
   print_this_as_int( (double) 2.2);
  return 0;
}
```

콤파일결과는 다음과 같다.

```
Is an integer value 1
Is an integer value 2
```

주의: 대부분의 콤파일러들은 이 경우에 정확도상실을 표시하는 경고를 내보낸다.

6.9.1 애매성

다중정의된 이름을 사용하면 콤파일러가 해석할수 없는 애매한 경우가 있을수 있다. 다음과 같은 원형을 가진 두 함수들의 경우를 고찰하자.

```
void print_this (const int);
void print_this (const double);
```

코드렬이 다음과 같은 경우에 콤파일러는 다중정의된 이 함수에 대하여 어느것을 호출해야 하는지 모르게 된다.

```
print_this ( ( short ) 2 );                    // 애매성
print_this ( ( long double ) 3.3 );            // 애매성
```

콤파일러는 어느 함수를 호출할지 모르겠다는것을 가리키는 오유통보문을 내보낸다.

print_this(const int)
or print_this(const double)

이러한 정합처리(matching process)는 클라스의 메쏘드들에서도 일어 난다.

6.9.2 요구하지 않은 형변환

우의 실례에서 보는바와 같이 함수선언에 파라메터를 정합시키는 문제에는 형들사이의 변환문제가 들어 있다. 두 수중에서 더 큰 수를 돌려 주는 함수인 경우에는 형변환이 요구되지 않는다. 우에서 서술한 정합방법에서는 함수호출에서 사용된 값들을 함수명세부에서 선언된 형으로 변환한다.

실례로

```
double larger (const double, const double);

double larger (const double first, const double second)
{
  return first > second ? first : second;
}
```

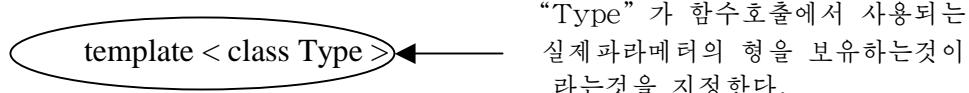
와 같이 정의된 함수 larger에 대하여 코드

```
larger ( 1, 4 )
```

는 double 형인 4를 돌려 준다. 실제파라메터들인 1과 4는 함수 larger가 호출될수 있도록 double 형직접값으로 형변환된다.

6.10 함수본보기

함수본보기(function template)는 모든 함수들에 대한 명세부이다. 함수본보기 명세부는 콤파일러가 함수호출에서 사용되는 실제파라메터에 따라 해당한 함수를 발생시키기 위한 본보기로서 사용할수 있는 범용함수들을 정의한다.

그림 6-5 는 larger 함수를 통하여 본보기함수의 구성요소들을 보여 준다.

```
template < class Type >   ← "Type"가 함수호출에서 사용되는
                             실제파라메터의 형을 보유하는것이
                             라는것을 지정한다.

Type larger( const Type first, const Type second )
{
    return first > second ? first : second;
}
```

그림 6-5. 본보기함수의 구성요소

이 선언은 함수에서 사용되는 실제파라메터의 형이 본보기지정자에서 선언되는 기호이름으로 주어 진다는것을 제외하고 일반함수선언과 같다. 지정자 template <class Type>는 Type 가 함수에 넘겨 지는 실제파라메터의 형을 보유한다는것을 가리킨다.

6.10.1 사용실례

다음의 실례프로그람은 함수본보기 larger 를 사용한다.

```
int main ( )
{
    std::cout << larger( 5,  4 )         << "\n " ;
    std::cout << larger(9.99,  4.64 )  << "\n " ;
    return 0;
}
```

함수본보기 larger 에 대한 코드를 콤파일하고 실행하면 결과는 다음과 같다.

```
5
9.99
```

6.10.2 함수본보기이름을 다중정의하기

두개 또는 세개의 수들중 최대값을 돌려 주는 두개의 함수본보기들은 다음과 같다.

```
template  <class Type>
Type larger (const Type first, const Type second)
{
```

```
  return first > second ? first : second;
}

template <class Type>
Type larger (const Type first, const Type second, const Type third)
{
  return larger (larger(first, second), third);
}
```

주의: 다중정의된 본보기함수 larger에 대한 파라메터들은 반드시 같은 형이여야 한다.
연산자 >는 반드시 두 파라메터들에 대하여 정의되여야 한다.
함수본보기에는 원형이 필요 없다.

우의 본보기함수들을 사용하여 다음의 코드를 작성할수 있다.

```
int  main ( )
{
 std::cout << larger (5, 4)                  << "\n";
 std::cout << larger (10, 30, 20)          << "\n";
 std::cout << larger (9.99, 4.64, 3.14) << "\n";
 std::cout << larger ('M', 'A', 'S')  << "\n";
 return 0;
}
```

결과는 다음과 같다.

```
5
30
9.99
S
```

알림

```
 std::cout << larger (4, 5.9)         <<  " \n ";
 std::cout << larger ('A', 66)      <<  " \n ";
```

우와 같이 콤파일된 코드에 대해서는 파라메터들의 형이 서로 다르므로 콤파일시 두개의 오유가 나온다. 이것을 해결하기 위해 다음과 같이 새 함수를 정의한다.

```
template <class Type1, class Type2>
Type1 larger_2g  (const Type1 first, const Type2 second)
{
     return first > (Type1) second ? first : (Type1) second;
}
```

다음의 시험프로그람으로 콤파일될 때

```
int main( )
{
  std::cout << larger_2g( 4, 5.9 )      <<  "\n ";
  std::cout << larger_2g( 'A', 66 )     <<  "\n ";
  return 0;
}
```

결과는 다음과 같다.

```
5
B
```

그러나 이렇게 항상 요구하지는 않을것이다.

주의: 이 본보기함수가 앞의 본보기함수 larger 와 결합된다면 서로 다른 함수이름들을 가져야 한다. 그것은 콤파일러가 어느 본보기함수를 사용해야 하는지 해석할수 없기때문이다.
함수결과형은 첫번째 실제파라메터의 형과 같다.

6.11 함수정합순서(다중정의된 함수)

함수정의와 함수호출을 정합시키기 위하여 C++에는 세개의 명확한 정합방법들이 있다. 즉

① 함수와 파라메터사이에 정합을 정확히 할수 있다면 이 정의를 사용하시오. 그렇지 않으면
② 함수본보기를 사용하는 함수와 그 파라메터들사이에 정합할수 있는가 보시오. 그렇지 않으면
③ 다중정의분석법을 리용하여 함수와 그 파라메터들사이에 정합을 할수 있는지 보시오.

애매성은 둘이상의 정합이 정합방법 ①에 의하여 진행되는 경우 생기게 된다.

6.11.1 다중정의분석법

다중정의분석법에는 다음의 처리들이 들어 있다.

- 함수의 매 파라메터에 대하여 이 파라메터와 정합될수 있는 다중정의함수들을 모두 찾는다.
- 모든 파라메터들에 대하여 정합을 할수 있는 함수정의가 적어도 하나 있게 된다. 만일 있다면 이것은 사용할수 있는 함수정의이고 그렇지 않으면 애매성오유가 발생된것으로 생각한다.

6.12 자체평가

- 프로그람을 안전하게 작성하는 관점에서 볼 때 값에 의한 실제파라메터넘기기의 우점은 무엇인가?
- 함수가 실행의 결과로서 값을 돌려 줄수 있는 경우 왜 실제파라메터에 대하여 참조에 의한 호출이 요구되는가?
- 함수원형이란 무엇이며 왜 필요한가?
- 함수이름의 다중정의는 왜 사용되는가?
- 프로그람에서 다중정의된 이름들의 우결함은 무엇인가?
- C++는 왜 유효범위해결연산자를 요구하는가?
- 재귀함수는 왜 내부전개(inline)를 할수 없는가?
- 함수선언에서 void의 목적은 무엇인가?
- 원형 void interesting()을 가지는 함수를 프로그람에서 리용하는것이 어떤 의의를 가지는가?
- 일반함수를 쓰는것보다 본보기화된 함수를 쓰는것의 우결함은 무엇인가?
- 어떤 경우에 일반함수가 아니라 본보기화된 함수를 사용하는가?
- 다중정의분석법이란 무엇인가?

6.13 련 습

다음의 함수들을 구성하시오.

- 산수연산
  arithmetic 함수를 쓰시오. 선언은 다음과 같다.

```
int arithmetic (int operand1, char op, int operand2);
```

이 함수는 여기서 두개의 형식파라메터 operand1 과 operand2 사이에 연산자 op 를 수행한 결과를 넘겨 준다. 실례로

```
std::cout<< “ the sum      of 1 and 2 is ” << arithmetic(1,   ‘+’ , 2);
std::cout<< “the product  of 2 and 3 is ” << arithmetic(2,   ‘*’ , 3);
```

- 산수연산본보기
  arithmetic 함수를 본보기화된 함수로 다시 쓰시오.

다음의 프로그람들을 작성하시오.

- 간단한 탁상수판
  연산자우선순위가 없는 탁상수판을 실행하는 프로그람을 쓰시오.
  실례로 공식 $2 + 3 \times 4$ 가 입력된다면 결과는 20 이다.
- 탁상수판
  탁상수판을 실현하는 프로그람을 작성하시오.
  실례로 공식 $2 + 3 \times 4$ 가 입력된다면 결과는 14 이다.

7 분할콤파일

이 장에서는 C++프로그람을 독립적으로 콤파일될수 있는 단위들로 가르는 방법에 대하여 서술한다. 이 분할콤파일(separate compilation)은 이미 존재하는 프로그람의 구성요소들을 재사용할수 있는 능력을 훨씬 증대시킨다.

7.1 소 개

큰 프로그람에서는 원천코드를 쉽게 조작하기 위하여 프로그람에서 리용되는 클라스들의 원천코드들을 개별적인 파일(separate file)들로 분할한다. 일반적으로 매 클라스마다 두개의 명백한 파일들이 있다.

- 클라스명세부를 포함하고 있는 파일.
- 클라스실현부를 포함하고 있는 파일.

클라스의 명세부는 #include 지령을 리용하여 그 클라스를 사용하는 파일에 포함된다. 일반적으로 이 파일이름에 그 클라스이름과 확장자 .h 를 붙인다. 실례로 Account 클라스의 명세부를 포함하는 파일이름은 Account.h 이며 그 내용은 다음과 같다.

```
#ifndef CLASS_ACCOUNT_SPEC
#define CLASS_ACCOUNT_SPEC

class Account {
public:
    Account( );
    float Account_balance( ) const;              // 잔고를 돌려 준다
    float withdraw ( const float );              // 구좌로부터 출금
    void deposit ( const float );                // 구좌에 저금
    void set_min_balance( const float );         // 최소잔고를 설정
private:
    float the_balance;                           // 현재잔고
    float the_min_balance;                       // 최소잔고
};

#endif
```

주의: 이 명세부코드가 우연히 한 파일에 두번 포함되는것을 막기 위하여 이 코드의 앞뒤에 다음의 전처리지령을 준다.

#ifndef ACCOUNT_SPEC	전처리기호 ACCOUNT_SPEC 가 존재하지 않는다면 그 다음본문을 콤파일한다.
#define ACCOUNT_SPEC	전처리기호 ACCOUNT_SPEC 를 정의한다.
#endif	코드의 조건적포함을 끌마친다.

이것들은 일반 if 명령문과 아주 류사하게 동작한다. 차이점은 다만 그것들이 콤파일시에 실행된다는것이다.
메쏘드 account_balance 는 검토자인데 이 메쏘드뒤에 const 예약어를 붙여야 한다. const 를 사용하면 형식파라메터들을 읽기만 한다.

마찬가지로 Account 클라스의 실현부도 다른 파일에 둔다. 약속에 의해 실현부에는 클라스이름과 확장자 .cpp 를 가진 이름이 붙는다. 이것도 앞뒤에 조건부콤파일지령들을 쓸수 있다. 실례로 Account 클라스의 실현부를 포함하는 파일의 이름은 Account.cpp 이며 그 내용은 다음과 같다.

```
#ifndef CLASS_ACCOUNT_IMP
#define CLASS_ACCOUNT_IMP

# include "Account.h"

Account::Account()
{
    the_balance = the_min_balance = 0.00;
}

float Account::account_balance() const
{
    return the_balance ;
}

 float account :: withdraw(const float money)
{
    if(the_balance - money >= the_min_balance)
    {
        the_balance = the_balance - money;
        return money;
    } else {
        return 0.00;
    }
 }

void Account::deposit (const float money)
{
    the_balance  = the_balance + money;
}

void Account::set_min_balance(const float money)
```

```
{
    the_min_balance = money;
}
#endif
```

주의: 전처리지령 # include "Account.h"를 사용하여 Account 클래스의 명세부를 포함시킨다.

Account 클래스의 시험프로그람은 다음과 같다.

```
# include <iostream>
# include <iomanip>
# include "Account.h"
int main ( )
{
  std::cout << std::setiosflags ( std::ios::fixed ) ;          // x.y 형식
  std::cout << std::setiosflags ( std::ios::showpoint ) ;      // 0.10
  std::cout << std::setprecision ( 2 ) ;                       // 소수부 2 자리
  Account mike;
  float obtained;
  std::cout << "Account balance  = " << mike.account_balance( ) << "\n" ;

  mike.set_min_balance ( -100.00 );
  std::cout << " Overdraft        = " << 100.00 << " \n " ;
  obtained = mike.withdraw ( 20.00 );
  std::cout << " Money withdrawn  = " << obtained << " \n " ;
  std::cout << "Account balance  = " << mike.account_balance ( ) << "\n " ;

  mike.deposit( 100.00 );
  std::cout << Money deposited   = " << 100.00 << " \n " ;
  std::cout << "Account balance  = " << mike.account_balance ( ) << "\n " ;

  obtained = mike.withdraw( 20.00 );
  std::cout << "money withdrawn  = " << obtained << "\n " ;
  std::cout << "Account balance  = " << mike.account_balance( ) << "\n " ;
  return 0;
}
```

주의: 전처리지령 # include "account.h"를 사용하여 account 클래스의 명세부를 포함시킨다. 그러나 실현부코드는 개별적으로 콤파일되기때문에 포함되지 않는다.

7.2 실제콤파일

콤파일처리의 정확한 세부는 콤퓨터들마다 서로 다르지만 g++(The Free Software Foundations C++compiler)를 사용하면 다음과 같이 될것이다.

지령	설명문
g++ -c Acount.cpp	실현부코드를 목적코드파일 Account.o 로 콤파일한다.
g++ -c test.cpp	시험프로그람을 목적코드파일 test.o 로 콤파일한다.
g++ test.o Account.o	시험프로그람을 위한 목적코드와 Account 클라스를 위한 실현부코드를 함께 련결하여 실행가능한 파일(executable image)로 만든다.

통합대상과제개발체계(integrated project development system)에서 사용자는 대상과제를 이루는 모든 C++원천프로그람들을 그 체계에 알린 다음 체계가 최소한의 재콤파일을 사용하여 설계를 재구축하게 한다.

7.3 분할콤파일과 inline 지령

분할콤파일을 진행하고 내부전개기능을 쓰려면 이 함수들은 반드시 클라스의 명세부에 포함되여야 한다. 이것은 콤파일러가 어느 코드를 내부전개(inline)해야 하는지 알아야 하기때문이다.

내부전개된 함수들이 개별적으로 콤파일된 실현부에서 정의되면 콤파일전반이 련결시에 오유를 발생시키게 된다. 그 리유는 다음과 같다. 함수가 개별적으로 콤파일되는 실현부에서 inline 으로 지정되였기때문에 이 함수들에 대하여 코드를 발생시키지 못한다. 따라서 이 메쏘드들을 사용하는 기본프로그람들을 콤파일할 때 콤파일러는 단순히 련결시에 실행되는 메쏘드에 대하여서만 참조를 놓는다. 이러한 메쏘드본체가 없기때문에 그 콤파일처리는 련결시에 오유를 발생시키게 된다.

이 해결책은 클라스의 명세부에 함수의 inline 실현부를 놓는것이다. 실례로 메쏘드 account_balance 와 클라스구축자를 내부전개하려면 클라스의 명세부는 다음과 같이 써야 할것이다.

```
#ifndef  CLASS_ACCOUNT_SPEC
#define CLASS_ACCOUNT_SPEC

class Account  {
public:
   Account ( );
   float  account_balance( )   const;                  // 잔고를 돌려 준다
   float withdraw ( const  float );                    // 구좌로부터 출금
```

```
    void  deposit ( const  float );                          // 구좌에 저금
   void set_min_balance( const  float );                     // 최소잔고를 설정한다
   private:
      float  the_balance;                                    // 현재잔고
      float  the_min_balance;                                // 최소잔고

// inline 부분들은 여기에 놓여야 한다

inline Account :: Account ( )
{
   the_balance  =  0.00;
}

inline float Account :: account_balance ( )  const
{
  return the_balance;
}

#endif
```

클라스의 실현부는 구축자(constructor)와 메쏘드 account_balance 코드를 뺀 과정을 제외하고는 같을것이다.

7.4 개인구좌관리프로그람의 재고찰

5.6 에서 서술한 개인구좌관리프로그람은 그의 주프로그람의 한 부분으로서 모든 입력과 출력을 진행하는 프로그람이다. 클라스 TUI 를 입출력을 진행하도록 작성하면 이 프로그람의 기본론리에서 입출력공정을 떼여 낼수 있다. TUI 클라스의 책임은 다음과 같다.

메쏘드	책임
menu	사용자에게 표시될 안내서를 설정한다. 안내서항목은 문자렬로 서술된다.
event	TUI 의 사용자에 의해 선택되는 안내서항목을 돌려 준다.
message	사용자에게 통보문을 표시한다.
dialog	사용자로부터 응답을 요구한다.

C++명세부는 다음과 같다.

```
#ifndef   CLASS_TUI_SPEC
#define   CLASS_TUI_SPEC
# include <string>

class TUI
```

```
{
public:
  enum  Menu_item { M_NONE, M_1, M_2, M_3, M_4, M_5, M_6 };
   TUI( );
   void  menu (std::string  =   " " , std::string  =   " " , std::string  =   " " ,
                  std::string  =   " " , std::string  =   " " , std::string  =   " " );
    Menu_item event ( );
    void message (std::string);
    void dialogue (std::string, float& );
    void dialogue (std::string, int& );
protected:
    void  display_menu_item (std ::string,  std::string);
private:
    std::string  the_men_1,  the_men_2,  the_men_3;
    std::string  the_men_4,   the_men_5,   the_men_6;
};

#endif
```

주의: protected 부는 11.5 에서 서술한다. 클라스명세부의 보호부(protected)령역들에는 클라스의 다른 메쏘드들에 의하여 리용되는 메쏘드들이 들어 있는데 이것들은 그 클라스의 밖에서 볼수 없다.
string 클라스의 구체례를 서술하는 표준클라스 std::string 을 사용하면 문자렬비교와 값주기에서 세련된 문자렬조작을 할수 있다. 부록 4 에 이 클라스성원들을 보여 준다.

실례로 TUI 의 구체례가

```
TUI  screen;
```

로 선언된 경우 안내서체계를

```
[a]   Print
[b]   Calculate

Input   selection:
```

라고 설정하기 위해 코드를 다음과 같이 쓴다.

```
 screen.menu (   "`Print  " ,    " Calculate  "   );
```

이 안내서에 대한 사용자의 응답은 event함수에 의해 유도된다. event함수는 선택된 안내서항목을 표시하는 렬거부를 돌려 준다.

실례로 사용자가 항목 [b]를 선택했다면 다음의 코드는 표식 M_2와 결합된다.

```
switch  ( screen.event ( ) )
{
    case  TUI :: M_1:                                          // 인쇄

    case  TUI :: M_2:                                          // 계산
}
```

주의: 선택된 안내서항목은 안내서항목 1 에 대해 렬거 M_1, 안내서항목 2 에 대해 M_2 를 가리킨다.
유효범위해결연산자 ::는 M_1 과 M_2 가 TUI 클라스의 성원이라는것을 가리키는데 쓰인다.

7.4.1 문자렬흐름

C++입출력클라스들을 사용하면 자료를 출력장치에 쓸수도 있고 기억기에 쓸수도 있다. ostrstream 클라스의 구체례는 cout 객체와 같은 객체이다. 실례로 다음의 코드는 memory 객체에 지구의 직경을 쓴다.

```
std ::ostrstream  memory;                                  // 문자렬흐름
long  earth_diameter  =12756;                              // 키로메터로

memory  << earth_diameter << `\0`;                         //기억기에 쓰기
```

기억기에 보관된 문자들의 마감을 표시하기 위해 memory 객체에 기호 '\ 0' 을 쓴다. 메쏘드 str()는 기억기에 쓴 자료를 표시하는 C++문자렬을 넘겨 준다.

실례로 earth_diameter 객체의 긴 옹근수를 문자렬 number 로 변환하기 위하여 다음의 코드를 사용한다.

```
# include  <iostream>                                      // 기본입출력흐름
# include  <strstream>                                     // 문자렬흐름클라스

int main( )
{
    std::ostrstream memory;                                // 문자렬흐름
    long  earth_diameter =12756;                           // 키로메터로

    memory <<earth _diameter <<`\0`;                       // 기억기에 쓰기

    std::string   number = memory.str ( );                 // 문자렬에 대입

    std::cout <<  "Earth`s  diameter is  "  <<
            number <<  " kilometers  "  <<  "\n" ;
     return 0;
}
```

콤파일하고 실행하면 결과는 다음과 같다.

```
Earth`s  diameter  is  12756  kilometres
```

알림

str 메쏘드에 의해 넘어 가는 C++문자렬을 보관하기 위해 사용된 기억구역은 체계에 돌려 지지 않는다. 결과 코드가 실행될 때마다 기억구역이 줄어 들게 된다.

7.4.2 종합서술

프로그람작성자는 출력해야 할 본문을 파라메터로 가지는 message 메쏘드를 사용하여 통보문을 표시할수 있다. 마찬가지로 프로그람작성자는 류점수를 되돌리는 dialog 메쏘드를 다중정의하여 사용자와 대화를 할수 있다. TUI 는 일반적으로 류점수나 옹근수를 요구하는 대화들만 지원한다.

마일을 키로메터로 변환하는 다음의 프로그람에서 TUI 클라스를 사용하여 입력과 출력을 조작한다.

```
# include <iostream>                                   // 기본입출력흐름
# include <iomanip>                                    // 입출력조작자
# include <strstream>                                  // 문자렬흐름클라스
# include  " TUI.h  "

int main ( )
{
  TUI  screen;                                        // 대화화면
   float  miles;                                      // 마일입력
   const  double  M_TO_K =1.609344;                   // 변환

   screen.dialogue(  "Miles  : " ,  miles );            // 대화

   std :: ostrstream  text;                            // 문자렬흐름
    text << std::setiosflags (std::ios::fixed);          // float 형
    text <<std::setiosflags(std::ios::showpoint);
    text <<std::setprecision(2);
    text <<  "Equals  : "  <<miles  *  M_TO_K
          <<  "kilometers  "  << `\0`;

    screen.message ( text.str ( )  );                    // 결과
    return 0;
}
```

주의: 키로메터수를 문자렬로 쉽게 변환하기 위해 문자렬흐름을 사용한다.

TUI 클라스를 결합하여 콤파일하면 다음의 대화방식결과가 나온다.

```
Miles    : 50.0
Equals  : 80.47  kilometres
```

7.4.3 개인구좌관리프로그람의 실현

대화방식의 개인구좌관리프로그람은 다음과 같다.

```
[a]  Deposit
[b]  Withdraw
[c]  Balance
[d]  Quit
Input  seclection :   c
Balance  = 4.55
```

주의: Quit 항목은 프로그람이 차례로 탈퇴되게 한다.

Account 와 TUI 클라스를 사용하여 개인구좌관리프로그람을 실현하는 프로그람은 다음과 같다.

```
# include <iostream>                 // 기초입출력흐름
# include <iomanip>                  // 입출력조작자
# include <strstream>                // 문자렬흐름클라스
# include  "Account.h "              // Account 클라스명세부
# include  "TUI.h"                   // TUI 클라스명세부

int main ( )
{
  Account  mine ;                    // 나의 구좌
  TUI  screen;                       // 대화화면
  float  amount;                     // 금액

  screen.menu(  "Deposit  " ,  "Withdraw   " ,   "Balance   " ,   "Quit  " );
```

개별적인 요구들은 switch 명령문을 사용하여 처리된다.

```
while  (  true  )
{
   switch (  screen.event()  )
   {
```

저금안내서항목이 선택될 때 요구을 처리하는 코드는 다음과 같다.

```
case  TUI :: M_1:
    screen .dialogue(  "Amount to deposit  ",  amount);
    if  (amount >= 0.0)
    {
       mine.deposit(amount);
    } else {
        screen.message(   "Amount  must  be  positive  " )
    }
    break;
```

출금처리코드는 다음과 같다.

```
case  TUI :: M_2:
    screen.dialogue (  "Amount  to  withdraw" , amount );
   if (amount >= 0.0)
   {
       float get = mine.withdraw(amount);
       if (get <= 0.0)
       screen.message(  "Sorry not enough funds" );
    } else {
       screen.message( "Amount must be positive" );
    }
    break;
```

잔고질문처리코드는 표시되는 본문을 얻기 위하여 문자렬흐름을 사용한다.

```
case  TUI :: M_3:
  {
    std::ostrstream text;                                       // str 흐름
    text  <<std::setiosflags(std::ios::fixed);                  // float 형
    text  <<std::setiosflags(std::ios::showpoint);
    text  <<std::setprecision ( 2 );
    text<< "Balance = " <<mine.account_balance( )<<`\0`;  // "Bal…"
    screen.message( text.str( )  );
}
break;
```

```
case  TUI :: M_4:
        return 0;
  }
 }
 return 0;
}
```

7.4.4 TUI 클라스의 실현부

TUI 클라스의 실현부에서 구축자는 안내서항목을 빈 문자렬로 설정한다. 그러므로 사용자가 menu 메쏘드를 호출하지 못하는 경우에도 코드는 계속 기능을 수행하게 된다.

```
#ifndef   CLASS_TUI_IMP
#define   CLASS_TUI_IMP

# include  " TUI.h "

TUI :: TUI( )
{
  the_men_1  =  the_men_2   =   the_men_3   =
  the_men_4  =  the_men_5   =   the_ men_6  =  "  " ;
}
```

menu 메쏘드는 다음번 현시를 위해 안내서항목들을 기록한다.

```
void TUI ::menu(std::string m1, std::string m2, std::string m3,
                std::stirng m4, std::string m5, std::string m6)
{
    the_men_1 = m1 ;   the_men_2  =  m2 ;    the_men_3  =  m3 ;
    the_men_4 = m4 ;   the_men_5  =  m5 ;    the_men_6  =  m6;        // 이름기억
}
```

안내서항목들을 표시한후에 event 메쏘드는 사용자가 안내서항목중 한개를 선택할것을 요구한다. 이 코드는 사용자가 안내서에 없는 항목(빈 문자렬)을 선택하지 않도록 작성되여 있다.

```
TUI :: Menu_item TUI :: event( )
{
   Menu_item choice = M_NONE;
   while  ( choice == M_NONE )
   {
     display_menu_item ( " [a]   " ,  the_men_1 );                    // 첫번째 안내항목
     display_menu_item ( " [b]   " ,  the_men_2 );                    // 두번째…
     display_menu_item ( " [c]   " ,  the_men_3 );
     display_menu_item ( " [d]   " ,  the_men_4 );
     display_menu_item ( " [e]   " ,  the_men_5 );
     display_menu_item ( " [f]   " ,  the_men_6 );
     char selection;
     cout <<  "Input selection : " , std::cin >>selection;
     switch (selection)
     {
       case  `a`  : case `A`:
```

```
        if(the_men_1  != " ") choice = M_1;
        break;
      case   `b` : case `B`:
         if(the_men_2 != " ") choice = M_2;
         break;
      case `c`  : case  `C`:
         if(the_men_3 != " ") choice = M_3;
         break;
       case `d` : case `D`:
         if (the_men_4 != " ") choice = M_4;
         break;
        case `e` : case `E`:
          if (the_men_5 != " ") choice = M_5;
          break;
        case `f `: case `F`:
          if (the_men_6!= " ") choice = M_6;
          break;
        default:
          break;
    }
    if (choice == M_NONE )
        message ( "Invalid response " );
  }
return choice;                                                // 사용자선택
}
```

보호부성원함수 display_menu 는 빈 안내서항목은 절대로 표시하지 않는다. 프로그람작성자는 이 클라스를 사용하여 일부 안내서항목들만 표시할수도 있다.

```
void TUI ::display_menu_item(std::string prompt, std::string name)
{
 if (name != " " )    //std::string 이 있다면
  {
     std::cout <<prompt <<name<<  "\n" << "\n" ;
   }
}
```

메쏘드 message 는 본문통보문을 출력장치에 표시한다.

```
void TUI:: message (std :: string  mes )
{
    std :: cout  << mes <<   "\n"<< "\n" ;                     // 사용자에게 통지
}
```

메쏘드 dialogue 는 류점수에도 혹은 옹근수에도 다 응답할수 있는 두개의 실현부를 가진다.

```
void TUI :: dialogue( std :: string mes, float &  answer )
{
  std :: cout << mes  <<  " : " ;                    // 사용자입력요구
  std :: cin   >> answer ;                           // 사용자응답읽기
  std :: cout <<  "\n" ;
}
void TUI :: dialogue ( std :: string mes, int &  answer )
{
  std :: cout << mes <<  " : " ;                     // 사용자입력요구
  std :: cin   >> answer;                            // 사용자응답읽기
  std :: cout <<  "\n " ;
}
#endif
```

7.4.5 종합서술

우의 프로그람은 지령선콤파일러 g++에 들어 있는 다음의 지령을 써서 콤파일될수 있다.

지령	설명문
g++ -c Account.cpp	실현부코드를 목적코드파일 Account.0 으로 콤파일한다.
g++ -c TUI.cpp	실현부코드를 목적코드파일 TUI.0 으로 콤파일한다.
g++ -c main.cpp	주프로그람을 콤파일한다.
g++ main.o Account.o TUI.o	Account 와 TUI 클라스의 실현부코드를 주프로그람과 련결하여 실행파일을 만든다.

두개의 클라스 Account 와 TUI 가 주프로그람과 결합되여 콤파일되고 실행될때 만들어 지는 대화방식의 실례는 다음과 같다.

```
[a]   Deposit
[b]   Withdraw
[c]   Balance
[d]   Quit
Input  selection :   a
Amount to deposit  :   10.00
```

```
[a]    Deposit
[b]    Withdraw
[c]    Balance
[d]    Quit
Input selection :    b
Amount to deposit  :   5.45
[a]    Deposit
[b]    Withdraw
[c]    Balance
[d]    Quit
Input selection :    c
Balance  =  4.55
```

7.5 자체평가

- 어떤 팀의 한 성원으로 작업할 때 클라스실현부를 개별적으로 콤파일하고 팀외 다른 성원들이 그의 명세부를 읽기만 할수 있게 하는것의 우점은 무엇인가?
- 큰 프로그람을 개별적으로 콤파일되는 각이한 파일들로 분할하자면 어떻게 해야 하는가?
- 어떤 팀의 한 성원으로 작업할 때 개별적인 파일들에 대한 접근권을 누가 가지게 하겠는가를 조종하는것이 중요하다. 왜 그런가?
- 왜 프로그람작성자가 클라스를 개별적으로 콤파일할 때 내부전개기능을 특별히 써야 하는가?

7.6 련 습

- 개인구좌관리프로그람을 실례로 하여 자기가 작성한 프로그람을 명세부파일과 실현부파일로 가르시오. 그리고 분할콤파일을 리용하여 이 프로그람을 콤파일하시오.

8 배 렬

이 장에서는 같은 객체들에 대하여 저준위집합을 실현하는 배렬구조에 대하여 서술한다. 경험이 적은 프로그람작성자들은 C++배렬을 리용할 때 많은 난관에 부닥칠수 있다. 그러나 표준본보기서고(Standard Template Library)에서 제공된 고준위구조들을 사용하여 이러한 부족점들을 없앨수 있다.

8.1 배 렬

대부분의 언어들에서와 마찬가지로 C++는 프로그람작성자들이 같은 항목의 구체례가 여러개 있을 때 그것을 배렬로서 기억시키고 조작할수 있도록 한다. 그러나 배렬은 얼핏 봐서는 잘 알수 없으며 경험이 적은 프로그람작성자들에게 있어서 이것은 오유를 일으킬수 있는 원인으로 된다.

5개의 옹근수형객체에 대한 배렬선언은 다음과 같다.

```
const int   NUMBER = 5;
int   room[NUMBER];
```

배렬은 항상 0 번 요소에서 시작되므로 5 개 요소로 된 배렬 room 은 0 번부터 4 번까지의 요소를 가진다. 개별적요소들은 배렬의 n 번째 요소를 선택하는 첨수(subscript)를 사용하여 접근한다. 실례로 배렬 room 의 2 번 요소에 값 15 를 설정하는 값주기명령문은 다음과 같다.

```
room[2] = 15;
```

다음의 명령문들은 방의 면적값들을 배렬 room의 개별적요소들에 기억시킨다.

```
room[0] = 20;   room[1] = 30;   room[2] = 15;   room[3] = 40;   room[4] = 10;
```

우의 명령문들을 실행한후 배렬 room의 내용들은 다음과 같이 된다.

배렬 room의 내용들	**배렬 room의 개별적요소들의 첨수**
20 | 30 | 15 | 40 | 10	0: 20 | 1: 30 | 2: 15 | 3: 40 | 4: 10

배렬이 선언될 때 여러 요소들을 어떤 값으로 초기화할수 있다. 실례로 배렬 room안에 있는 요소들을 초기값 20, 30, 15, 40, 10으로 설정하기 위하여 다음과 같이 선언한다.

```
int room[NUMBER] = { 20, 30, 15, 40, 10 };
```

이 선언은 배렬의 0번요소를 20, 1번 요소를 30, 2번 요소를 15 등으로 설정한다.

주의: 초기값을 너무 많이 주면 오유로 된다. 일부 요소들에 초기값을 주고 그 나머지 요소들은 int()로 설정하여 초기값을 0으로 할수 있다.

8.2 배렬의 리용

다음의 실례프로그람에서는 5개 방의 총 면적을 계산한다. 개별적방들의 면적값은 배렬 room안에 들어 있다.

```
#include <iostream>

int main()
{
    const int NUMBER = 5;
    int room[NUMBER] = { 20, 30, 15, 40, 10};

    std::cout << "Total area of all " << NUMBER << "rooms = ";

    int total = 0;
    for ( int i=0; i<NUMBER; i++ )
        {
            total = total + room[i];
        }
    std::cout << total << "Square metres"<< "\n";
    return 0;
}
```

우의 프로그람을 콤파일하고 실행하면 다음의 결과를 얻는다.

```
Total area of all 5 rooms = 115 Square metres
```

8.3 배렬의 첨수검사

많은 프로그람언어들은 배렬첨수가 유효한가를 검사한다. 첨수가 유효첨수값범위를 벗어 났다면 항상 실행시에 오유통보가 발생되며 프로그람이 완료된다. C++에서는 배렬경계에 접근하기 위하여 리용되는 배렬한계가 유효한가를 확인하는 실행시검사가 없다. 이것은 콤파일된 C프로그람의 코드를 빠르고도 간결하게 만들려는 요구로부터 나왔다. 만일 우의 프로그람을

```
int main( )
{
          const int NUMBER = 5;
           int room[NUMBER]={ 20, 30, 15, 40, 10 };
           std::cout <<  "Total area of all  "  << NUMBER <<  "  rooms =  ";
           int total = 0;
           for ( int i=0; i<=NUMBER; i++ )     // <--  틀림  ... i가  5로  될수  있다.
           {
                     total = total + room[i];
           }
           std::cout << total <<  "  Square metres"  <<  "\n";
           return 0;
}
```

와 같이 쓰면 요소 room[5]에 대한 접근이 진행되며 이것은 정의되지 않은 임의의 값을 준다. 즉 배렬 room이 끝난 다음위치의 내용이 total에 더해 진다.

주의: 잘못은 순환을 0부터 4까지가 아니라 5까지 진행한것이다.

이런 형태의 오유는 보통 프로그람에서 발견하기 매우 힘들다. 이 절에서 알아둘것은 C++에서 배렬에 접근하는 코드를 심중히 생각하고 작성해야 한다는것이다. 이 문제에 대한 해결방법은 20.2와 25.4에서 서술한다. 배렬의 접근을 빠르고 간결하게 하기 위해 C나 C++배렬에는 자체서술(self-describing)이 없으며 따라서 배렬에는 포함하는 총 요소수에 대한 정보가 들어 있지 않다.

8.4 다차원배렬

다음의 표는 선언형태와 다차원배렬의 매 성원들에 어떻게 접근되는가를 보여준다.

선언	개념	빗선 친 부분의 요소에 접근한다	기억기안의 물리적표시
int vector[4];	1	vector[2]	
int table[3][4];	2 1	table[1][2]	

선언	개념	빗선 친 부분의 요소에 접근한다	기억기안의 물리적표시
int cube[2][3][4];	1 2 3	cube[0][1][2]	

주의: 기억기에서 요소들에 차례로 접근할 때 맨 오른쪽의 첨수가 가장 빨리 변한다.

페지식기계에서 큰 배렬에 효률적으로 접근하자면 맨 안쪽 순환에서는 맨 오른쪽의 첨수부터 변화시켜야 한다. 그렇지 않으면 안쪽 순환고리에서는 배렬의 요소들에 분산적으로 접근하게 된다.

8.5 함수의 1차원배렬파라메터

실제파라메터가 배렬일 때에는 호출된 함수에 참조에 의하여 넘겨 진다. 이것은 배렬전체로서 넘겨 지는것이 아니라 그 배렬에 대한 참조가 넘겨 진다는것을 의미한다. 이렇게 하면 배렬의 참조(하나의 기계단어)만이 넘겨 지기때문에 배렬전체를 복사하는것보다 기억공간을 줄이며 시간도 단축한다. 그러나 함수의 형식파라메터로 배렬을 쓴다면(그것을 const로 선언하지 않은 경우) 실제파라메터로 넘어 온 배렬을 변경시키게 된다는것을 의미한다.

다음의 함수 sum은 옹근수배렬에서 요소들의 합을 돌려 준다. C++배렬에는 자체서술이 없으므로 배렬의 크기를 sum함수에 넘겨 주어야 한다. sum함수의 실현부는 다음과 같다.

```
int sum( int vec[], int size )
{
        int total = 0;
        for ( int i=0; i<size; i++ )
        {
                total = total + vec[i];
        }
        return total;
}
```

주의: 형식파라메터 vec의 선언에서 크기가 정의되지 않았다. 이것은 C++배렬에 자체서술이 없기때문이다. 이런 방식으로 선언한 배렬을 열린 배렬(open array)이라고 부른다.

우의 함수 sum을 리용하여 몇개의 방에 있는 성원들의 수를 계산하는 프로그람은 다음과 같다.

```
int main()
{
const int ROOMS = 4;
int people[ROOMS] = { 2, 3, 1, 3 };

std::cout << "Total number of people in the rooms is "
          << sum( people, ROOMS ) << "\n";
return 0;
}
```

우의 프로그람을 콤파일하고 실행하면 다음과 같이 된다.

```
Total number of people in the rooms is 9
```

8.6 함수의 다차원배렬파라메터

C++에서는 n차원배렬을 함수에 실제파라메터로 넘길 때 그의 마지막으로부터 n−1개의 한계값들만을 지적한다. 이러한 크기정보는 콤파일러가 함수본체안에서 배렬의 매개 요소를 어떻게 호출하겠는가를 계산하기 위하여 리용한다. 한계값들은 콤파일시에 상수로 서술되여야 하므로 이것은 함수의 일반성에 제한을 준다.

실례로 임의의 크기를 가진 행렬(2차원배렬)을 조작하는 범용함수를 작성하기는 어려운것이다. 한가지 방법은 저준위의 기초지적자를 사용하는것이다(지적자들은 17장에서 설명한다).

배렬의 한계값들은 실행시에 검사되지 않으므로 첫 첨수의 한계값은 생략한다. 콤파일러는 다른 첨수의 한계값들로부터 배렬요소위치를 계산할수 있다.

실례로 2D(2차원)배렬의 내용을 인쇄하고 요소들을 왼쪽에서 오른쪽으로 뒤집기하는 프로그람은 처음에 상수선언으로 배렬의 크기를 정의한다.

```
#include <iostream>

const int ROW       = 3;
const int COLUMN    = 4;
```

다음 왼쪽에서 오른쪽까지 배렬요소들을 인쇄하고 뒤집기하는 두 함수에 대한 원형들을 정의한다. C++에서 첫번째 차원의 크기는 배렬의 한계값검사가 없으므로 정의하지 않는다. 그러나 그뒤의 차원들에 대한 크기들은 콤파일러가 배렬의 선택된 요소의 위치를 계산할수 있도록 정의되여야 한다.

```
void print( const int table [ ][COLUMN] );
void flip_left_right( int table [ ][COLUMN] );
```

처음 프로그람은 배렬을 형성하고 2차원배렬을 두번 인쇄하는데 두번째의것은 배렬요소들이 왼쪽에서 오른쪽으로 뒤집어 진것이다.

```
int main()
{
        int numbers[ROW][COLUMN];

        for( int i=0; i<ROW; i++ )               // ROW와 COLUMN자리표로
        for( int j=0; j<COLUMN;   j++ )
        number[i][j] = i*10+j;                   //요소들을 배치한다

        std::cout << "Original array" << "\n";
        print( numbers );                        // 2차원배렬수들을 인쇄
        flip_left_right( numbers );              // 왼쪽에서 오른쪽으로 뒤집기
        std::cout << "Flipped array" << "\n";
        print( number );                         // 2차원배렬수들을 인쇄
        return 0;
}
```

print함수의 실현부를 아래에 보여 주었다.

```
void print( const int table[][COLUMN] )
{
     for( int i=0; i<ROW; i++ )                  // 매행에 대하여
     {
        for( int j=0; j<COLUMN/2; j++ )          // 행의 매렬에 대하여
        {
            std::cout << table[i][j] << "\t";    // 요소를 인쇄
        }                                        // for순환의 끝
        cout <<  "\n";                           // 행바꾸기
     }                                           // for순환의 끝
 }
```

함수 flip_left_right의 실현부는 다음과 같다.

```
void flip_left_right( int table[][COLUMN] )
{
   for( int i=0; i<ROW; i++ )                    // 매행에 대하여
   {
     for( int j=0; j<COLUMN/2; j++ )             // 렬에서
     {                                           // 매 요소에 대하여
         int temp = table[i][j];                 //
        table[i][j] = table[i][COLUMN-j-1];      // 반사시키기
        table[i][COLUMN-j-1] = temp;
      }                                          // for순환의 끝
   }                                             // for순환의 끝
}
```

주의: 2차원배렬의 첫번째 차원은 열린 첨수(open subscript)이므로 요구되지 않는다. 배렬이 파라메터로 넘어 갈 때는 그의 참조가 넘어 간다. 형식배렬파라메터를 const로 정의하여 함수본체에서 형식배렬에 대한 쓰기를 막을수 있다.

우의 프로그람을 콤파일하고 실행하면 다음의 결과가 나온다.

```
Original array
0	1	2	3
10	11	12	13
20	21	22	23
Flipped array
3	2	1	0
13	12	11	10
23	22	21	20
```

주의: C++에서는 코드작성시 개별적차원들의 한계값 혹은 전체 배렬의 한계값을 초과하는 경우 오유가 발생된다. 그러므로 이런 현상을 방지하는 기구가 요구된다. 그 해결방도를 20.2와 25.4에서 서술하였다.

8.6.1 배렬의 초기화

배렬은 선언할 때 다음과 같이 초기화할수 있다.

```
int vec[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

주의: 만일 모든 요소들에 초기값을 주지 않으면 나머지요소들은 Type()로 초기화된다. 여기서 type는 그 배렬의 형이다. 이것은 그 형에 따르는 0값을 돌려 준다.

첨수는 배렬의 크기가 모든 요소들을 포함할수 있는 경우에 생략될수 있다.

문자배렬을 초기화하는 빠른 방법은 개별적문자상수값을 주지 않고 C++문자렬을 리용하는것이다.

```
char name[] = "Brighton";
```

이때 문자렬끝기호가 들어 가는데 우의 선언문은 아래와 같이 쓸수도 있다.

```
char name[] = { 'B', 'r', 'i', 'g', 'h', 't', 'o', 'n', '\0' };
```

주의: 너무 많은 값으로 초기화하는 경우에 오유가 발생한다.

8.6.2 더 높은 차원의 배렬초기화

이것도 같은 방법으로 하는데 다만 매 차원에 대한 항목을 묶어 주기 위하여 { }들을 리용한다.

```
int table[2][3]      = { {1, 2, 3}, {4, 5, 6} };
int cube[2][3][4] = { { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} },
                        { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} } };
```

주의: 안에 있는 {}들은 생략될수도 있다.

8.7 객체배렬의 초기화

구축자를 가지고 있는 객체도 int와 같은 기본형의 배렬에서와 같은 방법으로 초기화할수 있다(8.6과 15.9를 참고하시오).

8.8 OX유희

OX유희는 3×3살창들에서 진행한다. 매 선수는 교대로 빈 4각형에 각각 표식 O와 X를 덧붙인다. 유희는 한 선수가 자기의 표식 3개를 대각선, 수직, 수평중 어느 한줄에 놓았을 때 이긴것으로 된다. 만일 빈 4각형을 다 채우게 되면 유희는 비긴것으로 된다(그림 8-1).

<table>
<tr><th>X의 첫번째 수</th><th>O의 첫번째 수</th><th>X의 첫번째 수</th><th>O의 첫번째 수</th></tr>
<tr>
<td><table><tr><td>x</td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr></table></td>
<td><table><tr><td>x</td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td>O</td><td></td></tr></table></td>
<td><table><tr><td>x</td><td></td><td>x</td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td>o</td><td></td></tr></table></td>
<td><table><tr><td>x</td><td>o</td><td>x</td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td>o</td><td></td></tr></table></td>
</tr>
<tr><th>X의 세번째 수</th><th>O의 세번째 수</th><th>X의 네번째 수</th><th></th></tr>
<tr>
<td><table><tr><td>x</td><td>o</td><td>x</td></tr><tr><td></td><td>x</td><td></td></tr><tr><td></td><td>o</td><td></td></tr></table></td>
<td><table><tr><td>x</td><td>O</td><td>X</td></tr><tr><td></td><td>X</td><td></td></tr><tr><td>o</td><td>O</td><td></td></tr></table></td>
<td><table><tr><td>x</td><td>o</td><td>x</td></tr><tr><td></td><td>x</td><td></td></tr><tr><td>o</td><td>o</td><td>x</td></tr></table></td>
<td>보는바와 같이 처음으로 진행하는 선수가 유리하다.</td>
</tr>
</table>

그림 8-1. O X의 유희

유희는 판과 그우에 X와 O를 교대로 놓는 두 선수가 한다. 이러한 유희의 개념적모형을 그림 8-2에 보여 주었다.

판

x 선수

o 선수

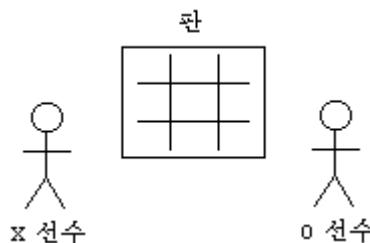



그림 8-2. O와 X유희의 개념적모형

이 유희의 현재상태를 보여 주는 프로그람은 Board클라스로 작성할수 있다. Board클라스의 책임들은 다음과 같다.

메쏘드	책 임
add	판에 선수의 표식을 추가한다. 선수가 쓸 수는 1부터 9범위의 수와 한 문자의 표식으로 지정한다.
valid	현재 쓴 수가 정확하다면 참을 돌린다. 이 메쏘드는 쓴 수가 1부터 9범위내에서 있는가, 지적된 세포가 빈것인가를 검사한다.
view	판의 매 쪼각들의 내용을 표현한 문자렬을 돌려 준다. 실례로 우에서 진행한 유희의 마지막상태는 문자렬 "XOX X OOX"로 표현된다.
situation	OX판의 현재상태를 돌려 준다.

이 클라스에 대한 C++명세부는 다음과 같다.

```
#ifndef CLASS_BOARD_SPEC
#define CLASS_BOARD_SPEC

#include <string>

class Board
{
public:
   enum Game_result { WON, DRAWN, PLAYABLE };
   Board();
   bool valid( const int pos ) const;              // 쓴 수가 정확한가
   void add( const int pos, const char piece ); // 판에 추가
   std::string view() const;                       // 유희상태문자렬을 돌려 주기
   Game_result situation() const;                  // 유희의 상태
private:
   static const int SIZE_TTT=9;                    // 9이여야 한다
   char the_sqrs[SIZE_TTT];                        // 유희살창
   int the_moves;                                  // 쓴 수
};

#endif
```

8.8.1 문자렬에서 첨수의 리용

문자렬은 매 문자들에 접근할수 있도록 첨수화될수 있다. 사실상 문자렬은 0부터 length()−1까지의 한계값을 가진 배렬로서 취급될수 있다. 실례로 문자렬을 소문자로 변환하는 함수는 다음과 같다.

```
void to_lower_case( std::string& str )
{
	for ( int i=0; i<str.length(); i++ )
	{
		str[i] = tolower( str[i] );
	}
}
```

주의: 이 코드는 그 원형이 머리부파일 <ctype>에 정의되여 있는 서고함수 tolower를 리용한다. 부록 3에서는 머리부파일 <ctype>에 정의되여 있는 함수들을 보여 준다.

8.8.2 기억루실이 없는 문자렬흐름구체례

앞에서 문자렬흐름을 사용할 때 요구되는 기억기가 할당되였으나 해방되지 않는다는것을 지적하였다. 구체례를 참조하기 위하여 문자렬흐름구체례에 기억기를 할당함으로써 이러한 기억루실은 피할수 있다. 실례로 최대로 100개의 문자를 보유할수 있는 문자렬흐름구체례인 text의 만들기는 다음과 같은 선언을 리용한다.

```
const int MAX_BUF = 100;               // text의 최대크기
char buf[MAX_BUF];                     // 씌여 진 text를 보유한다
ostrstream text( buf, MAX_BUF);        // text는 문자렬흐름이다
```

정보는 다음과 같은 표준입출력추출연산자를 사용하여 기억기안에 씌여 진다.

```
long earth_diameter = 12756;           // 키로메터로

text << “Earth’s diameter is ” <<
        earth_diameter << “ kilomtres” << ‘\0’;

std::string message = text.str();
std::cout << message << “\n”;
```

주의: 문자‘\0’은 기억기에서 문자들의 마감에 추가되여야 한다.
메쏘드 str()는 message를 초기화하는 C++문자렬을 전송한다.

우의 프로그람을 콤파일하고 실행하면 다음의 결과를 내보낸다.

```
Earth’s diameter is 12756 kilometres
```

8.8.3 OX유희구동프로그람

주프로그람코드는 두 선수로부터 쓴 수(move)를 도출해 낸다. 유희는 어느 한 선수가 이길 때까지 혹은 더 다른 수가 없을 때까지 계속된다. 대화(interaction)는 구동프로그람과 입출력을 분리시키기 위해 TUI클라스구체례를 사용한다. 우에서 보

여 준 Board클라스와 TUI클라스를 리용한 구동프로그람의 코드는 다음과 같다.

```
#include <iostream>
#include <string>

// Board클라스와 TUI클라스

std::string as_text_pic( std::string );                    // 판의 본문쪼각
int main()
{
    char player;                                           // X 혹은 O선수
    Board oxo;                                             // 판의 구체레
    Board::Game_result game_is = Board::PLAYABLE;   // 판의 상태
    TUI screen;

    player = ‘X’;                                          // 첫 선수
    while ( game_is = = Board::PLAYABLE )      // 유희를 진행할수 있는 동안
    {
        int move;
        std::string who = std::string(“Player ”)+player;  // 선수 X 혹은 O는
        screen.dialogue(who + “ enter move”, move);  // 정확한
        if ( oxo.valid( move ) )                           // 쓸 수를 의뢰한다
        {
            oxo.add( move, player );                       // 판에 추가하기
            screen.message( as_text_pic(oxo.view() ) );    // 판을 보여 주기
            game_is = oxo.situation();                     // 유희의 상태
            switch ( game_is )
            {
              case Board::WON        :                     // 이김
                screen.message( who+ “wins”);
                break;
              case Board::DRAWN      :                     // 비김
                screen.message( “ It’s a draw”);
                break;
              case Board::PLAYABLE :
                switch ( player )                       // 유희할수 있는 상태
                {
                case ‘X’: player = ‘O’; break;       // ‘X’를 ‘O’로
                case ‘O’: player= ‘X’; break;        // ‘O’를 ‘X’로
                }
                break;
            }
        } else {
          screen.message( “Move invalid” );          // 잘못 쓴 수
```

```
        }
    }
    screen.message( “  ” ); screen.message( “  ” );
    return 0;
}
```

주의: Board클라스에 정의된 렬거형 Board_state의 이름 붙은 상수에 접근하기 위하여 유효범위해결연산자 ::를 사용한다. 또한 문자렬은 매 문자들에 접근할수 있도록 첨수화될수 있다.

함수 as_text_pic는 OX판을 출력장치에 본문으로 표시하는데 알맞는 형태로 표현한 문자렬을 돌려 준다. 함수입력은 view메쏘드가 돌려 주는 문자렬인데 OX판의 매 4각형의 내용을 준다. 새 문자렬은 판의 기호표시형태에 알맞는 기호들로서 만들어 진다.

메쏘드 view가 돌려주는 문자렬에서 매 문자의 위치를 보여주는 판	위치(메쏘드 view가 돌려 주는 문자렬에서)	판을 그리기 위하여 추가되는 본문
1 ｜ 2 ｜ 3	1,2,4,5,7,8	“ ｜ ”;
-----------------		
4 ｜ 5 ｜ 6	3,6	“\n-----------------\n”;
-----------------		
7 ｜ 8 ｜ 9	9	“\n”;

이렇게 하면 OX판을 문자에 의하여 2차원적으로 볼수 있게 하는 문자렬을 만들수 있다. 함수 as_text_pic는 다음과 같이 정의된다.

```
std::string as_text_pic( std::string rep )
{
    std::string res = “  ”;
    for( size_t i=1; i<=rep.length(); i++ )
    {
        res += rep[i-1];                          // 계수기를 추가한다
        switch ( i )                              // 계수기추가후
        {
            case 3 :
            case 6 :                              // 행분리기호를 추가한다
                    res += “\n-----------------\n”;
                    break;
            case 9 :                              // 행바꾸기기호를 추가한다
                    res += “\n”;
```

```
                    break;
            case 1 : case 2 : case 4 :
            case 5 : case 7 : case 8 :                    // 렬분리기호를 추가한다
                    res +=  "|";
                    break;
        }
    }
    return res;                                           // 판의 본문그림을 돌려 준다
}
```

주의: size_t는 머리부파일 <stddef.h>에 정의되여 있다.

두 선수사이의 대표적인 유희과정에 대한 출력을 아래에 보여 주었다.

X의 첫번째 수	O의 첫번째 수	X의 두번째 수	O의 두번째 수
x | | ------------------ | | ------------------ | |	x | | ------------------ | | ------------------ | o |	x | | x ------------------ | | ------------------ | o |	x | o | x ------------------ | | ------------------ | o |
X의 세번째 수	**O의 세번째 수**	**X의 네번째 수**	
x | o | x ------------------ | x | ------------------ | o |	x | o | x ------------------ | x | ------------------ o | o |	x | o | x ------------------ | x | ------------------ o | o | x	보는바와 같이 처음으로 진행하는 선수가 유리하다

8.8.4 Board클라스실현부

Board클라스실현부에서 구축자는 판을 모든 공간들의 정의된 상태로 설정한다.

```
#ifndef CLASS_BOARD_IMP
#define CLASS_BOARD_IMP
#include "Board.h"
Board::Board()
{
        for ( int i=0; i<SIZE_TTT; i++ )
        {
                the_sqrs[i] =  ' ';
        }
        the_moves = 0;
}
```

함수 valid는 선택된 4각형이 앞서 진행한 상대편에 의하여 점령된것이 아닐 때 true를 돌려 준다.

```
bool Board::valid( const int pos ) const
{
    return (pos >= 1 && pos <= SIZE_TTT) && the_sqrs[pos-1] == ' ' ;
}
```

주의: C++는 불완전평가를 사용하므로 &&연산자의 왼쪽은 그의 오른쪽이 LHS가 참일 때에만 평가된다.
그러나 만일 &연산자가 사용되면 두쪽이 다 평가된다.

함수 add는 판에 선수의 표식을 추가한다. 이 메쏘드는 수를 쓰는 상태가 valid 메쏘드에 의하여 확인된 조건에서만 사용된다.

```
void Board::add( const int pos, const char piece )
{
        the_sqrs[ pos-1 ] = piece;
        the_moves++;
}
```

메쏘드 view는 OX판의 현재상태를 돌려 준다.

```
std::string Board::view() const
{
        std::string res = " ";
        for ( int i=1; i<=SIZE_TTT; i++ )
        {
                res +=the_sqrs[i-1];
        }
        return res;
}
```

판의 현재상태를 결정하기 위하여 매개의 이길수 있는 줄을 검사한다. 이것은 이길수 있는 매개 줄에 대한 모든 자리표값들을 2차원배렬 win_lines에 보관함으로써 실현된다. 2차원배렬에서 첫번째 첨수는 이길수 있는 줄을 선택하며 두번째 첨수는 이길수 있는 줄에 대한 3개의 자리표값들을 선택한다.

모든 표식이 'X' 혹은 'O'로 되였는가를 매개 이긴 줄에 대하여 검사한다. 만일 이긴 줄이 발견되지 않았다면 비김상태에 대한 검사를 진행한다.

```
Board::Game_result Board::situation() const
{
        const int WL = 8;                              // 이길수 있는 줄의 수
```

```
        const int LL = 3;                                          // 줄의 길이
        int win_lines[WL][LL] = { {0, 1, 2}, {3, 4, 5}, {6, 7, 8},
                                  {0, 3, 6}, {1, 4, 7}, {2, 5, 8},
                                  {0, 4, 8}, {2, 4, 6} };
        for ( int i=0; i<WL; i++ )                                 // 이길수 있는
        {                                                          // 매줄에 대하여
                char first_cell = th_sqrs[ win_lines[i][0] ];
                if ( first_cell != ' ' &&
                     first_cell == the_sqrs[ win_lines[i][1] ] &&
                     first_cell == the_sqrs[ win_lines[i][2] ] )
                   return WON;
        }
        if ( the_moves >= SIZE_TTT ) return DRAWN;

        return PLAYABLE;                                           // 유희 계속 진행
}
#endif
```

8.9 배렬을 리용한 탄창구조의 구축

탄창은 자료항목을 보관하고 검색하기 위하여 리용되는 구조이다. 자료항목들은 그 구조에 넣어 지며 추가된 순서의 반대로 검색된다. 이것은 보통 "선입후출"에 귀착한다. 이러한 처리과정을 그림 8-3에서 보여 주었다.

꺼내기(1) 꺼내기(2) 넣기( ) 넣기( )

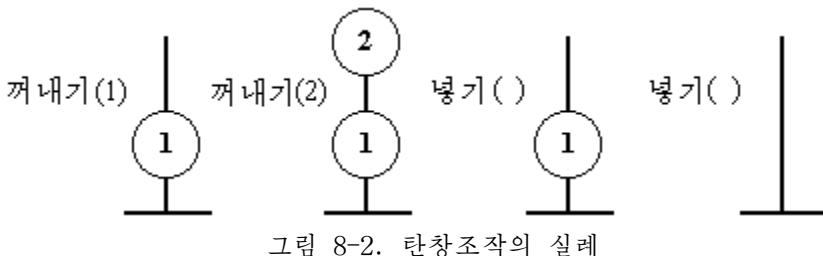



그림 8-2. 탄창조작의 실례

주의: C++함수가 호출될 때마다 그의 돌림주소가 탄창에 넣어 진다. 그다음 함수가 탈퇴될 때 탄창에서 돌림주소가 검색된다. 중첩되는 함수의 호출도 이런 방법으로 진행된다.
함수가 호출될 때마다 호출한 함수다음의 명령주소가 탄창에 넣어 진다. 함수가 탈퇴될 때 이 주소는 탄창에서 삭제되며 조종은 거기로 이행된다.

탄창클라스의 책임은 다음과 같다.

메쏘드	책 임
empty	탄창이 빈 경우 참을 돌려 준다.
pop	탄창의 꼭대기항목을 삭제한다. 돌림값은 없다.

메쏘드	책 임
push	탄창에 새로운 항목을 넣는다.
size	탄창에 있는 요소들의 수를 돌려 준다.
top	탄창에서 꼭대기에 있는 항목을 돌려 준다.

탄창을 표현한 클라스의 정의부분은 다음과 같다.

```
#ifndef CLASS_STACK_SPEC
#define CLASS_STACK_SPEC

#include <ctype.h>

class Stack {
public:
        Stack();
        bool empty() const;                  // 탄창이 비였는가를 보기
        size_t size() const;                 // 집합의 크기
        int top() const;                     // 꼭대기항목을 돌려 준다
        void push( const int );              // 탄창에 항목을 넣기
        void pop();                          // 탄창의 꼭대기항목을 꺼내기
private:
        static const int MAX_ELEMENTS=5;
        int the_elements[MAX_ELEMENTS];  // 탄창의 항목들
        int the_tos;                 // 탄창의 꼭대기항목을 가리키는 지적자
};

#endif
```

주의: size_t형은 머리부파일 <ctype.h>에 정의되여 있는데 탄창에 있는 배렬의 크기를 정의하는데 리용된다.

탄창은 마지막으로 추가된 항목의 첨수값을 보관하는 the_tos를 가진 옹근수형의 배렬로서 실현할수 있다.

클라스구축자는 탄창꼭대기항목의 첨수값을 표현하는 the_tos를 가진 정의된 상태에 탄창을 설정한다. −1은 탄창이 비였다는것을 의미한다.

```
#ifndef CLASS_STACK_IMP
#define CLASS_STACK_IMP
#include <stdexcept>

Stack::Stack()
{
        the_tos = −1;                        // 비였다
}
```

메쏘드 empty와 size는 탄창의 상태를 알아 보는데 리용된다. empty함수는 탄창이 비였는가를 판정하는데 리용된다.

```
bool Stack::empty() const
{
        return the_tos < 0;                         // 비였는가
}
```

메쏘드 size는 탄창에서 요소들의 수를 돌려 준다.

```
size_t Stack::size() const
{
        return the_tos+1;                           // 탄창의 요소들
}
```

메쏘드 top는 탄창의 꼭대기항목값을 돌려 준다. 이때 탄창안의 값은 변경되지 않는다. 이 함수를 실현하자면 적어도 한개 요소가 탄창에 있어야 한다. 만일 탄창에 요소들이 없다면 례외가 발생한다. 14장에서 클라스사용에서 조작상 오유에 대한 책임을 진 례외기구에 대해 더 구체적으로 설명한다.

례외는 조작상 오유처리에 대한 좋은 방법인데 례외환경에서만 리용된다. 례외가 발생될 때 조종은 례외조종자(handler)에로 이행된다. 이것은 보통 프로그람작성자가 탄창의 구체례에 통보문을 보내기 위하여 리용한다. 만일 이러한 준비가 이루어 지지 않았으면 프로그람은 치명적오유로 인정하고 중단된다. 례외클라스들은 머리부파일 <stdexcept.h>에 정의되여 있다.

```
int Stack::top() const
{
        if ( the_tos < 0 ) {                        // 탄창이 비였는가
                throw std::range_error(“Stack: underflow”);
        }
        return the_elements[ the_tos ];             // 탄창의 꼭대기항목을 지적
}
```

메쏘드 push는 탄창에 새 항목을 추가한다. 만일 가능하지 않으면 례외가 내보내(throw)진다.

```
void Stack::push( const int item )
{
        if ( the_tos>= MAX_ELEMENTS-1 ) {           // 범위를 벗어 났다면
                throw std::range_error(“Stack: overflow”);
        }
        the_elements[ ++the_tos] = item;            // 항목을 추가
}
```

메쏘드 pop는 탄창에서 꼭대기항목을 삭제한다. 또한 조작의 가능성에 대한 일관한 검사를 진행한다.

```
void Stack::pop()
{
        if ( the_tos < 0 ) {                        // 탄창이 비였는가
                throw std::range_error(“Stack: underflow”);
        }
        the_tos--;                                  // 항목을 제거
}
```

이 클라스는 배렬 elements의 한계값을 벗어 나는 접근을 하지 않으므로 안전하다. 그러나 다음의 경우는 례외가 발생한다.

- 빈 탄창에서 항목을 삭제하는 경우.
- 탄창에 항목이 다 찼을 때 다른 항목을 추가하는 경우.
- 빈 탄창에서 꼭대기항목에 접근한 경우.

8.9.1 례외기구에 대한 소개

례외는 throw구조가 내보내며 catch구조가 포착한다. try블로크는 이러한 구조들에 대한 용기를 형성한다. 실례로 아래의 코드부분에서는 례외 runtime_error(“Too many items”)가 내보내지고 포착된다.

```
int main()
{
        try{

                if ( data_values > MAXDATA_VALUES )
                        throw std::runtime_error(“Too many items”);
        }
        catch ( std::runtime_error & err )
        {
                cout << “Fail: ” << err.what() << “\n”;
        }
        return 0;
}
```

주의: 메쏘드 what는 클라스상수 range_error(“Too many items”)를 창조할 때 리용한 문자렬 “Too many items”를 보낸다.
례외기구에 대해서는 14장에서, 이름공간(namespace)에 대하여서는 13장에서 서술한다.

8.9.2 종합서술

다음의 코드는 간단한 자료를 가지고 클라스 Stack에 대한 검사를 진행한다.

```
#include <iostream>
int main()
{
   Stack numbers;
   char ch;
   while ( std::cin >> ch, !std::cin.eof() )
   {
      try
      {
         switch ( ch )
         {
            case ‘+’ :                         // 탄창에 항목넣기
            {
                  int num; std::cin >> num; numbers.push(num);
                  break;
             }
            case ‘-’ :                         // 탄창에서 항목꺼내기
            {
               int num = numbers.top()
               std::cout << “Num = ” << num << “\n”;
               number.pop();
               break;
            }
         }
      }
      catch ( std::range_error & err )
      {
         cerr << “\n” << “Fail: ” << err.what() << “\n”;
      }
   }
   return 0;
}
```

탄창에서 무효조작이 진행될 때 례외 range_error가 내보내지며 이것을 례외조종자가 포착한다. 례외조종자는 오유흐름에 오유통보문을 인쇄하고 프로그람을 완료한다.

주의: 입력에서 공백은 무시된다.
례외가 발생되면 해당유효범위를 벗어 나는 객체를 해제하는 표준기구가 호출된다.

자료

```
+1 +2 +3 +4 - - - - -
```

를 가지고 실행시키면 다음과 같은 결과가 나온다.

```
Num = 4
Num = 3
Num = 2
Num = 1
Fail: Stack underlow
```

8.10 콤퓨터화된 은행체계

C++에서 배렬은 변수는 물론 객체들도 취급할수 있다. 실례로 아주 간단한 은행업무를 콤퓨터체계로 실현하기 위한 프로그람작성은 은행구좌에 대한 배렬을 요구한다. 이것을 실현하기 위하여 클라스 Bank는 다음의 책임들을 가진다.

메쏘드	책 임
account_balance	매 손님구좌에 대한 잔고를 돌려 준다.
Deposit	지적된 손님구좌에 돈을 저금한다.
last_account_no	마지막구좌번호를 돌려 준다.
set_min_balance	손님에 대한 초과출금의 한계를 설정한다.
statement_summary	구좌계산서에 대한 개요를 출력한다.
Valid	구좌번호가 정확한가를 검사한다.
Withdraw	손님구좌에서 가능하다면 돈을 출고한다.

Bank클라스의 명세부는 다음과 같다.

```
#ifndef CLASS_BANK_SPEC
#define CLASS_BANK_SPEC

class Bank {
public:
        Bank();
        ~Bank();
        float account_balance( const int ) const;           // 잔고
        float withdraw( const int, const float );           // 출금
        void deposit( const int, const float );             // 저금
        void set_min_balance( const int, const float );     // 최소잔고설정
```

```
        int last_account_no() const;                    // 마지막구좌번호
        void statement_summary( ostream&, const int ) const;
private:
        static const int MAX_CUSTOMERS=100;
        Account customer[MAX_CUSTOMERS];                // 손님구좌
};

#endif
```

주의: 프로그람은 배렬을 포함하므로 배렬에서 항목의 수 MAX_CUSTOMERS가 변경된다면 실현부코드는 다시 콤파일되여야 한다. 이러한 경우에 원천코드가 다시 리용될것이다. ~Bank()는 Bank클라스구체례가 완료될 때마다 호출되는 함수이다. 실례로 이 함수는 그것이 선언된 함수 혹은 블로크가 탈퇴될 때 호출된다. 이 함수를 해체자라고 부른다. 해체자에 대해서는 11.6에서 서술한다.

클라스배렬을 선언할 때 다음과 같은 형식으로 선언하였다면 클라스에는 파라메터가 없는 구축자가 있어야 한다.

```
Account customer[MAX_CUSTOMERS];                        //사용자구좌
```

주의: 우의 선언에서 Account에 대한 구축자는 MAX_CUSTOMERS개수만큼 호출된다.

클라스구체레들의 배렬을 지정값으로 초기화하는 방법에 대하여서는 15.9에서 보기로 한다. Bank클라스의 실현부는 다음과 같다.

```
#ifdef CLASS_BANK_IMP
#define CLASS_BANK_IMP
#include “Bank.h”
```

Bank클라스의 구축자와 해체자는 프로그람을 호출한 다음에 손님구좌에 있는 정보를 다시 기억시키고 보관한다. 구축자는 Bank구체례에 대한 기억기할당이 진행된후에 만들어 지며 해체자는 할당된 기억기가 체계에 넘겨 지기전에 호출된다. 여기에 적당한 코드를 추가하여 프로그람실행이 끝나기전에 은행업무의 내용을 보관할수 있다.

```
Bank::Bank()
{
        // 은행구좌를 설정하는 코드
}

Bank::~Bank()
{
        // 저금한 돈을 보관하는 코드
}
```

주의: 구축자와 해체자에 대해서는 11.6에서 보기로 한다.

메쏘드들인 account_balance와 withdraw, deposit, set_min_balance는 account클라스에서 련관된 메쏘드를 호출한다.

```
float Bank::account_balance( const int client ) const
{
        return customer[client].account_balance();
}
float Bank::withdraw( const int client, const float money )
{
        return customer[client].withdraw( money );
}
void Bank::deposit( const int client, const float money )
{
        customer[client].deposit( money );
}
void Bank::set_min_balance( const int client, const float money )
{

        customer[client].set_min_balance( money );
}
```

메쏘드 last_account_no는 마지막구좌의 번호를 돌려 준다. Bank클라스의 구체례 piggy에서 유효한 구좌번호들은 다음과 같이 된다.

0 .. piggy.last_account_no()

```
int Bank::last_account_no() const
{
        return MAX_CUSTOMERS-1;
}
```

메쏘드 statement_summary는 구좌잔고의 개요를 인쇄한다.

```
void Bank::statement_summary( std::ostream& s, const int client ) const
{
        s << "Bank statement summary -";
        s << " account number " << client << "\n";
        s << "£" << customer[client].account_balance();
        s << " on deposit" << "\n";
}
#endif
```

클라스배렬에서 메쏘드에 접근하는 구성요소를 그림 8-4에서 보여 주었다.

그림 8-4. 은행구좌를 호출할 때 구성요소들

8.10.1 종합서술

우의 Bank클라스를 사용하여 프로그람을 다음과 같이 쓸수 있다.

```
// Bank클라스 코드
int main()
{
        Bank piggy;
        float obtained;
        int customer = piggy.last_account_no();

        piggy.statement_summary( std::cout, customer );

        std::cout << "\n" << "Transaction Deposit   £100.00" << "\n";
        piggy.deposit( customer, 100.00 );
        piggy.statement_summary( std::cout, customer );

        std::cout << "\n" << "Transaction withdraw  £20.00"<< "\n"
        obtained = piggy.withdraw( customer, 20.00 );
        std::cout << "piggy Bank gives  £  << obtained << "\n";
        piggy.statement_summary( std::cout, customer );

        std::cout << "\n" << "Transaction Deposit  £50.00"<< "\n";
        piggy.deposit( customer, 50.00 );
        piggy.statement_summary( std::cout, customer );
}
```

우의 프로그람을 콤파일하고 실행하면 다음의 결과가 나온다.

```
Bank statement summary - account number 99
   £0 on deposit

Transaction Deposit  £100.00
Bank statement summary - account number 99
   £100 on deposit

Transaction withdraw  £20.00
piggy Bank gives  £20
```

```
Bank statement summary - account number 99
   £80 on deposit

Transaction Deposit  £50.00
Bank statement summary - account number 99
   £130 on deposit
```

주의: Account에 대한 구축자는 Bank클라스의 customer배렬의 매 요소에 대하여 호출된다. 마찬가지로 해체자는 Bank클라스의 구체례에 대하여 할당된 기억기가 유효범위에서 벗어 날 때 customer배렬의 매 요소에 대하여 호출된다.

8.11 부분관계

우의 실례에서 Account는 Bank의 부분(part_of)으로 된다. Bank클라스의 구체례는 Account클라스의 구체례들을 포함한다.

8.12 배렬과 문자렬

C나 C++에서 저준위문자렬은 특수표식문자 '\0'로 끝나는 문자배렬에 의하여 표현된다. 표식문자 '\0'은 0비트패턴(비트들이 모두 0인 바이트)에 의하여 표현된다. 이 표식문자는 문자렬끝을 결정 짓는데 리용된다. 실례로

```
char message[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

는 C++문자렬을 표시하는 message객체를 선언한다. 이것은 아래의 프로그람에서 보여 준것처럼 표준입출력기구를 사용하여 인쇄할수 있다.

```
int main()
{
        char message[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
        std::cout << message << "\n";
        return 0;
}
```

우의 프로그람을 콤파일하고 실행하면 다음의 결과가 나온다.

```
Hello
```

주의: message선언은 다음과 같이 쓸수도 있다.

```
char message[] = "Hello";
```

이 형식에서 표식문자 '\0'은 자동적으로 문자렬끝기호로 추가된다.

8.12.1 C++문자렬상수

문자렬상수는 특수표식문자 ‘\0’으로 끝나는 문자배렬에 의하여 표현된다. 실례로 아래의 프로그람에서

```
#include <iostream>

int main()
{
	std::cout << “Hello world” << “\n”;
	return 0;
}
```

문자렬 “Hello world”는 사실상 특수한 표식문자 ‘\0’으로 끝나는 문자배렬이다.

배렬의 끝기호로서 이러한 특수표식문자를 사용함으로써 그 배렬을 처리하는 쏘프트웨어는 그 문자렬안에 몇개의 문자가 들어 있는가를 결정할수 있다. 이것은 배렬의 0번째 요소부터 앞으로 나가면서 이 표식문자가 나타날 때까지 탐색하는 방법으로 결정된다.

이것은 앞의 6.6에서 본 std::string클라스를 리용한것과는 완전히 다르다. 그러나 문자렬상수 실례로 “Hello world”는 항상 이 방법으로 표시된다. string클라스는 문자렬상수를 리용할 때 배렬표시를 자기자체의 내부적표시로 변환한다.

8.12.2 C++문자렬을 사용한 실례

다음의 함수 write는 파라메터인 C++문자렬을 cout에로 인쇄한다.

```
void write( const char vec[] )
{
	int i = 0;
	while ( vec[i] != ‘\0’ )
	{
		std::cout << vec[i];
		i++;
	}
}
```

함수 write는 다음과 같이 리용할수 있다.

```
int main()
{
	write( “Hello world” );
	write( “\n” );
	return 0;
}
```

주의: “Hello world”는 함수 write에 참조로 넘겨 진 초기화된 문자배렬이다.

우의 프로그람을 콤파일하고 실행하면 다음의 결과가 나온다.

```
Hello world
```

8.12.3 C++문자렬을 리용할 때 좋은 점과 나쁜 점

문자렬을 표시하기 위하여 문자배렬을 사용하면 효률적인 코드를 작성할수 있지만 항상 위험이 동반된다. 일부 경우에 효률은 좀 떨어 지지만 문자렬을 표시하는 string클라스구체레를 리용하면 프로그람을 더 안전하고 편리하게 작성할수 있다. 다음의 표는 문자렬이 다음과 같이 선언되였을 때 그것들사이의 주요한 차이를 아래의 표에서 보여 주었다.

- 클라스 string
- 문자배렬 혹은 문자지적자(지적자의 개념에 대해서는 17장에서 론의된다)

	a 의 선언		
객체 a는	**Std:: string a**	**char a[10]**	**해설**
값주기될수 있다.	✓	✗	배렬은 값주기될수 없다.
함수에 참조로 넘겨 질수 있다.	✓	✓	배렬은 오직 참조로 넘겨 질수 있다.
함수에 값으로 넘겨 질수 있다.	✓	✗	배렬은 함수에 오직 참조로만 넘겨 질수 있다.
비교될수 있다.	✓	✗	배렬을 비교하면 그 요소들의 비교결과를 넘겨 주지 않는다[1].
C++문자렬의 문자로 값주기되거나 비교될수 있다.	✓	✗	배렬은 값주기나 비교를 하지 못한다[2].

주의: [1] 코드

```
char name1[] = “Mike”; char name2 = “Mike”;
if ( name1 = name2 ) cout << “names are equal” << “\n”;
```

에서는 배렬의 내용이 비교되지 않고 배렬의 물리적시작주소가 비교된다. 우의 실례에서 비교결과는 항상 거짓으로 된다. 17장에서 지적자리용에 대하여 더 구체적으로 설명한다.

[2] 비교연산자를 다중정의하고 그에 알맞는 구축자를 제공함으로써 string클라스는 C++문자렬과 string클라스구체레사이의 비교와 값주기를 실현한다. 15장에서는 표준연산자를 어떻게 다중정의하는가를 보여 준다.
배렬을 선언할 때 초기값을 줄수 있다.

8.13 사람이름과 주소를 관리하는 클라스

사람은 Person클라스구체레로 표현될수 있다. 이 클라스의 책임은 다음과 같다.

메쏘드	책 임
Person	사람의 이름과 주소를 설정한다.
address_line	주소의 n번째 행을 문자렬로 돌려 준다.
Lines_in_address	주소의 행수를 돌려 준다.
Name	사람의 이름을 문자렬로 돌려 준다.
set_address	사람의 이름과 주소에 대한 내용을 변경한다.

구축자와 메쏘드 set_address의 파라메터는 "Mike Smith/Brighton"형태의 문자렬인데 주소에서 새로운 행의 시작은 문자 '/'로 지적된다.
이 클라스에 대한 명세부는 다음과 같다.

```
#ifndef CLASS_PERSON_SPEC
#define CLASS_PERSON_SPEC
#include <string>

class Person {
public:
        Person( const std::string name = "Unknown" );
        void set_address( const std::string );          // 새로운 주소를 설정
        std::string name() const;                       // 이름을 돌려 준다
        int lines_in_address() const;                   // 주소의 행들
        std::string address_line( const int ) const;    // 주소의 n행
private:
        std::string the_details;                    // Person클라스의 세부내용
};
#endif
```

주의: 사람에 대한 정보들은 string클라스구체레로 보관된다. 문자렬은 배렬첨수연산을 리용하여 접근할수 있다. 그러나 첨수의 유효성은 검사하지 않는다.

Person클라스의 구축자와 메쏘드 set_address는 후에 검색할수 있도록 사람의 이름과 주소정보들을 보관한다.

```
#ifndef CLASS_PERSON_IMP
#define CLASS_PERSON_IMP
#include “Person.h”

Person::Person( const std::string details )
{
        set_address( details );
}

void Person::set_address( const std::string details )
{
        the_details = details;
}
```

메쏘드 name은 사람의 이름을 돌려 주는데 이것은 메쏘드 address_line을 사용하여 검색된다. 주소의 첫행이 사람의 이름이라는것은 하나의 요구조건이다.

```
std::string Person::name() const
{
        return address_line(1);
}
```

주의: name함수를 지정한 다음에 const를 사용하였다. 이것은 메쏘드가 클라스구체레의 어떤 항목도 수정하지 못한다는것을 가리킨다. const메쏘드는 오직 const클라스구체레우에서 호출할수 있는 유일한 함수이다.

메쏘드 lines_in_address는 주소의 전체 행수를 돌려 준다.

```
int Person::lines_in_address() const
{
        int lines = 1;
        for ( size_t i=0; i<the_details.length()-1; i++ )
        {
                if ( the_details[i] == ‘/’ ) lines++;
        }
        return lines;
}
#endif
```

메쏘드 address_line은 개인주소의 n번째 행을 전송한다. 이 함수는 이미 정의된 문자렬부분을 넘겨 주기 위하여 substr메쏘드를 사용한다.

substr메쏘드의 파라메터는 다음과 같다.

메쏘드	파라메터번호	형	서 술
substr	1	**const** size_t	부분의 시작위치
	2	**const** size_t	부분에서 문자들의 수

부록 4에서 표준C++클라스 string의 메쏘드들을 렬거한다.

```
std::string Person::address_line( const int line ) const
{
   const int last_char = the_details.length() - 1;
   int line_on = 1;
   for ( int i=0; i<last_char; i++ )
   {
      if ( line_on == line )
      {
            for ( int j=1; j<last_char; j++ )
            {
                  if ( the_details[j] == '/')
                  {
                      return the_details.substr( i, j-i );
                  }
            }
            return the_detail.substr( i, last_char-i+1 );
       }
       if ( the_details[i] == '/') line_on++;
   }
   return   "";
}
```

8.13.1 종합서술

```
//클라스 Person에 대한 코드

int main()
{
          Person me( "Mike Smith/University of Brighton/"
                          "School of Computing /Brighton" );
          for ( int i=1; i<=me.lines_in_address(); i++ )
          {
                    std::cout << me.address_line( i ) << "\n";
          }
          std::cout << "\n\n";
          return 0;
}
```

주의: 두개이상의 련속문자렬들은 콤파일러에 의하여 하나의 문자렬로 조립된다. 이런 방법으로 하나의 긴 문자렬을 더 품위 있고 읽기 쉽게 쓸수 있다.

우의 프로그람을 콤파일하고 실행하면 다음과 같은 결과가 나온다.

```
Mike Smith
University of Brighton
School of Computing
Brighton
```

8.14 자체평가

- **프**로그람작성오유를 발견하기 힘든 C++배렬을 왜 리용하는가?
- 배렬항목의 형에는 어떤 제한이 있는가?
- 배렬접근에서 련속된 기억기위치에 접근할 때 어느 첨수가 가장 빨리 변화되는가? 이것을 아는것이 왜 중요한가?
- 배렬이 가질수 있는 차원수에는 얼마만한 한계가 있는가?
- 객체배렬은 어떻게 선언되는가?
- 객체배렬들이 선언될 때 호출되는 구축자의 수는 얼마인가?

8.15 련 습

다음의 프로그람을 작성하시오

- OX유희를 콤퓨터선수와 진행
  선수에게 제안된 수를 돌려 주는 메쏘드를 Board클라스에 추가하시오.
- 도서관
  작은 학교도서관에 있는 책들의 상태를 관리하는 프로그람. 도서관에 있는 매책은 1~999범위의 수로서 부류기호(class mark)를 가진다. 둘이상의 같은 책들은 같은 부류기호를 가진다. 매 사람은
  - ✧ 도서관에서 나간 책을 검사할수 있다.
  - ✧ 대부한 책을 연기할수 있다.
  - ✧ 책의 현재상태를 물을수 있다.

  프로그람은 매일 우의 처리를 조종할수 있다. 게다가 도서관에 있는 책의 현재상태에 대한 개요를 제공한다. 실례로

```
Books in library 100
Books on loan   = 5
Books reserved = 2
Books on shelves = 93
```

참고: 책클라스를 만드시오. 책클라스의 책임은 다음과 같다.

메쏘드	책　임
loan	대부한 책을 표식한다.
missing	빠진 책을 표식한다.
reserved	연기된 책을 표식한다.
returned	서고에 책을 돌려 준다.
state	책의 상태를 돌려 준다. on_shelf, on_loan 등

9 본보기

본보기기구(template mechanism)는 클라스를 일반적으로 정의할수 있게 한다. 본보기클라스는 사실상 파라메터화된 형(parameterized type)이다. 프로그람작성자는 클라스의 구체례를 정의할 때 본보기클라스에서 어떤 형을 쓸것인가를 선택할수 있다. 본보기클라스를 만듦으로써 재리용코드령역을 상당히 증대시킬수 있다. 그러나 보통 본보기클라스를 만드는데는 더 많은 노력이 요구된다.

9.1 본보기클라스소개

Stack클라스코드(8.9에서 소개되였다.)는 옹근수형의 탄창에서만 리용된다. 프로그람작성자는 본보기클라스를 만들어 지적한 형으로 동작하는 클라스체를 만들수 있다. 본보기클라스를 파라메터화된 형이라고도 부른다. 실례로 본보기클라스 Stack의 명세부는 다음과 같다.

```
#ifndef CLASS_STACK_SPEC
#define CLASS_STACK_SPEC

#include <ctype.h>

template <class Type>
class Stack {
public:
        Stack();
        bool empty() const;                    // 탄창이 비였는가
        size_t size() const;                   // 집합의 크기
        const Type& top() const;               // 꼭대기항목을 돌려 준다
        Type& top();                           // 꼭대기항목을 돌려 준다
        void push( const Type );               // 탄창에 항목을 넣는다
        void pop();                            // 탄창에서 꼭대기항목을 꺼낸다
private:
        static const int MAX_ELEMENTS=5;
        Type the_elements[MAX_ELEMENTS]; // 탄창의 항목들
        int the_tos;                     // 탄창의 꼭대기항목을 가리키는 지적자
};
```

주의: 탄창에서 사용된 Type는 본보기머리부에 파라메터로서 서술된다.

메쏘드 top는 객체의 불필요한 복사를 피하기 위하여 집합의 항목에 대한 참조를 돌려 준다. 그러나 여기에는 이 메쏘드의 판본이 요구된다.

메쏘드서술	호출조건
Type& top( );	변이객체의 참조가 요구될 때
const Type& top() **const**;	상수객체의 참조가 요구될 때

아래의 코드는 다른 top메쏘드를 호출한다.

선　언	Top호출	리용되는 메쏘드
const Stack <**int**> c	c.top()	**const** Type& top() **const**;
Stack <**int**> m	m.top()	Type& top() **const**;

본보기클라스의 요소들을 그림 9-1에서 보여 주었다. 그림에서 일부 요소들은 생략되였다.

클라스에서 리용되는 형들의 이름을 지정한다

```
template < class Type >
class Stack {
public:
    stack();

 Type&  top();

private:

 Type   the_elements[MAX_ELEMENTS];
}
```

본보기파라메터를 리용한다

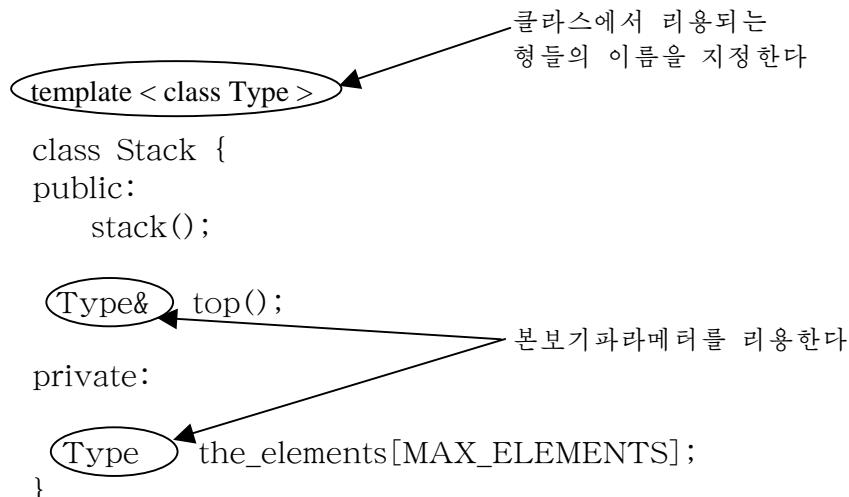



그림 9-1. 본보기클라스의 요소

탄창의 구체례는 다음과 같이 될수 있다.

```
Stack <int>     int_stack;          // 옹근수형탄창
Stack <char>    char_stack;         // 문자형탄창
```

주의: 파라메터화된 형은 여러형의 항목들로 구성된 탄창을 허용하지 않는다. 16장에서 이러한 여러형의 객체들을 다형성객체로 어떻게 실현하겠는가 하는 방법을 보여준다. 그러나 이것이 항상 요구되는것은 아니다.

앞에서 서술한 실현부들은 본보기를 지원하지 않는다. 이것은 마크로를 사용하여 실현할수 있다. 마크로를 사용하여 본보기효과를 내는 기구는 26.9에서 서술된다. 탄창의 실현부에서는 같은 방법으로 본보기구조체를 사용하는데 이것은 매 메쏘

드의 본체를 서술하기전에 지적하는 형파라메터이름이다.

```
#include <stdexcept>
template <class Type>
Stack<Type>::Stack()
{
        the_tos = -1;                           // 비였다
}
```

그림 9-2는 본보기클라스에서의 메쏘드실현부를 보여 준다.

```
template <class Type>          ← 클라스에서 리용되는
                                 형들의 이름을 지정한다
Stack<Type>:: Stack()          ← 이 구축자가 클라스 Stack
{                                에 속한다는것을 지정한다
   the_tos = -1;
}
```

그림 9-2. 본보기클라스에서 메쏘드의 실현부

메쏘드 empty와 size는 다음과 같다.

```
template <class Type>
bool Stack<Type>::empty() const
{
        return the_tos < 0;                     // 비였는가
}
template <class Type>
size_t Stack<Type>::size() const
{
        return the_tos+1;                       // 탄창의 요소들
}
```

메쏘드 top, push, pop는 다음과 같다.

```
template <class Type>
const Type& Stack<Type>::top() const
{
        if ( the_tos < 0 ) {                    // 탄창이 비였다면
                throw std::range_error ( "Stack: underflow" );
        }
        return the_elements[ the_tos ];         // 꼭대기항목
}
```

```
template <class Type>
Type& Stack<Type>::top()
{
        if ( the_tos < 0 ) {                        // 탄창이 비였다면
                throw std::range_error("Stack: underflow" );
        }
        return the_elements[ the_tos ];          // 꼭대기항목
}
```

주의: top메쏘드의 두 판본가운데서 하나는 고정탄창에서 리용되며 다른 하나는 비고정탄창에서 리용된다. 이 메쏘드들은 상수(읽기만 하는)객체의 참조를 돌려 주는 고정탄창과 변이(읽기/쓰기)객체의 참조를 돌려 주는 비고정탄창을 취급할 때 필요된다.

```
template <class Type>
void Stack<Type>::push( const Type item )
{
        if ( the_tos >= MAX_ELEMENTS-1 ) {          // 범위를 벗어 났다면
                throw std::range_error( "Stack: overflow" );
        }
        the_elements[ ++the_tos ] = item;                       // 항목추가
}
```

```
template <class Type>
void Stack<Type>::pop()
{
        if ( the_tos < 0 ) {                                    // 탄창이 비였다면
                throw std::range_error( "Stack: underflow" );
        }
        the_tos--;                                              // 제거
}
#endif
```

주의: 파라메터화된 클라스는 파라메터로서 많은 형을 가질수 있다. 실례로 클라스 Stack는 다음과 같은 2개의 파라메터를 가질수 있다.

```
template <class Type1, class Type2>
class Stack {
        ...
}
```

9.2 본보기클라스에서 다중파라메터

본보기함수와 마찬가지로 본보기클라스도 다중파라메터화된 형을 가질수 있다. 파라메터화된 형은 지적한 값의 형을 서술하는데 리용된다. 실례로 탄창클라스에서 탄창의 요소개수를 지적하기 위하여 사용된 배렬크기는 본보기에서 파라메터로 정의할수 있다. 이러한 클라스의 명세부는 다음과 같다.

```
#ifndef CLASS_STACK_SPEC
#define CLASS_STACK_SPEC
#include <ctype.h>
template <class Type, const int MAX_ELEMENTS=5>
class Stack {
public:
        Stack();
        bool empty() const;                     // 탄창이 비였는가
        size_t size() const;                    // 집합의 크기
        const Type& top() const;                // 꼭대기항목을 돌려 준다
        Type& top();                            // 꼭대기항목을 돌려 준다
        void push( const Type );                // 탄창에 항목을 넣기
        void pop();                             // 탄창에서 꼭대기항목을 꺼내기
private:
        Type the_elements[MAX_ELEMENTS];        // 탄창의 항목들
        int     the_tos;             // 탄창의 꼭대기항목을 가리키는 지적자
};
#endif
```

메쏘드 push의 실현부는 다음과 같다.

```
#include <stdexcept>
template <class Type, const int MAX_ELEMENTS>
void Stack<Type, MAX_ELEMENTS>::push( const Type item )
{
        if ( the_tos >= MAX_ELEMENTS-1 ) {      // 범위를 벗어 났다면
                throw std::range_error( "Stack: overflow" );
        }
        the_elements[ ++the_tos ] = item;               // 항목추가
}
```

주의: 다른 메쏘드들의 실현부도 이런 형식을 따른다.

Stack클라스의 구체례는 다음과 같이 선언할수 있다.

```
Stack <int,10> int_stack;        // 10개의 요소들을 가지는 옹근수형탄창
Stack <char> char_stack;         // 기정개수의 요소들을 가지는 문자형탄창
```

9.3 본보기클라스에서의 문제점

본보기클라스를 리용할 때 클라스작성자와 클라스사용자사이에 본보기클라스를 어떻게 리용해야 하는가 하는 약속이 없어도 된다. 많은 경우 본보기클라스에서 파라메터가 어떤 형으로 되든 상관이 없다. 그러나 실례로 산수계산식이 본보기클라스 실제파라메터의 구체례우에서 수행된다면 그 파라메터는 산수형이거나 리용된 산수연산자를 지원하는 형이여야 한다. 5.3.1에서 본 Account본보기클라스를 생각해 보자. 그의 명세부는 다음과 같다.

```
#ifndef CLASS_ACCOUNT_SPEC
#define CLASS_ACCOUNT_SPEC

template <class Type>
class Account {
public:
        Account();
        Type account_balance() const;               // 잔고를 돌려 준다
        Type withdraw( const Type );                // 구좌에서 출금한다
        void deposit( const Type );                 // 구좌에 저금한다
        void set_min_balance( const Type );         // 최소잔고를 설정한다
private:
        Type the_balance;                           // 현재잔고
        Type the_min_balance;                       // 최소잔고
};

#endif
```

클라스의 실현부는 다음과 같다.

```
#ifndef CLASS_ACCOUNT_IMP
#define CLASS_ACCOUNT_IMP

template <class Type>
Account <Type>::Account()
{
        the_balance = Type();                       // 령
        the_min_balance = Type();                   // 령
}
```

주의: 구축자는 지적된 형에 대한 령값을 돌려 준다. 즉 산수형인 경우에 그 형에 대한 령값을 돌려 준다. 실례로 int()는 옹근수형의 령값을 돌려 준다.

```
template <class Type>
Type Account<Type>::account_balance() const
{
        return the_balance;
}
template <class Type>
Type Account<Type>::withdraw( const Type money )
{
        if ( the_balance - money >= the_min_balance )
        {
                the_balance = the_balance - money;
                return money
        } else {
                return Type();                          // 령
        }
}
template <class Type>
void Account<Type>::deposit( const Type money )
{
        the_balance = the_balance + money;
}
template <class Type>
void Account<Type>::set_min_balance( const Type money )
{
        the_min_balance = money;
}
#endif
```

9.3.1 종합서술

우의 본보기클라스 Account를 리용하여 다음과 같은 코드를 쓸수 있다.

```
int main()
{
        Account <double> mike;
        double obtained;
        std::cout << std::setiosflags( std::ios::fixed );            // x.y형식
        std::cout << std::setiosflags( std::ios::showpoint);         // 0.10
        std::cout << std::setprecision(2);                           // 소수점아래 2자리
```

```
        std::cout << "Account balance = " << mike.account_balance() << "\n";
        mike.deposit(100.00);
        std::cout<< "Account balance = " << mike.account_balance() << "\n";
        obtained = mike.withdraw(20.00);
        std::cout << "Money withdrawn = " << obtained << "\n";
        std::cout << "Account balance = " << mike.account_balance() << "\n";
        mike.deposit(50.00);
        std::cout << "Account balance = " << mike.account_balance() <<
        "\n";
        return 0;
}
```

우의 프로그람을 콤파일하고 실행하면 다음과 같은 결과가 나온다.

```
Account balance   = 0.00
Account balance   = 100.00
Money withdrawn = 20.00
Account balance   = 80.00
Account balanc    = 130.00
```

또한 본보기클라스가 정확한 량을 취급하는 구좌를 주도록 하기 위하여 long형파라메터로 리용할수 있다. 이것은 단일한 화폐단위가 리용되는 경우에 쓰일수 있다.

```
int main()
{
        Account <long> mike;
        long obtained;
        std::cout << "Account balance = "<< mike.account_balance() << "\n";
        mike.deposit(10000);
        std::cout << "Account balance = "<< mike.account_balance() << "\n";
        obtained = mike.withdraw(2000);
        std::cout << "Mony withdrawn ="<< obtained << "\n";
        std::cout << "Account balance = "<< mike.account_balance() << "\n";
        mike.deposit(5000);
        std::cout << "Account balance = "<< mike.account_balance() << "\n";
        return 0;
}
```

우의 프로그람을 콤파일하고 실행하면 다음의 결과가 나온다.

```
Account balance = 0
Account balance = 10000
Money withdrawn = 2000
Account balance = 8000
Account balance = 13000
```

9.3.2 문제점

본보기클라스작성자는 파라메터형에 대한 제한조건을 서술할수 없다. 그러므로 프로그람을 아래와 같이 작성한 경우 틀린 결과가 나온다.

```
int main()
{
	Account <char> strange;				// 콤파일한다
	strange.deposit(97);
	std::cout << “mike Balance = ” << strange.account_balance() << “\n”;
	return 0;
}
```

우의 프로그람을 콤파일하고 실행하면 다음의 결과가 나온다.

```
mike Balance = a
```

Account클라스의 구체례를 다음과 같이 리용하는 경우 오유가 발생한다.

```
Account < Account<int> > wrong;
```

주의: >>연산자로 혼돈하지 않도록 하기 위하여 >와 >사이에 공백을 삽입하였다.

우의 프로그람은 콤파일되지 않으며 나타난 오유통보문도 도움을 주지 못한다. 그리고 Account <int>형의 구체례들사이에 +가 있는 곳에 오유가 있다고 가리킨다. 그러므로 본보기클라스문서에서 본보기클라스의 실제파라메터가 어떤 형으로 되여야 하는가를 잘 정의할 필요가 있다. 이로부터 본보기클라스리용자들은 구체례에서 실제파라메터로 리용될수 있는 형들의 제한을 잘 알아야 할것이다.

9.4 분할콤파일과 본보기클라스

본보기클라스는 리용될 때 클라스의 여러 구체례들에 대하여 유일하게 실행되여야 한다. 가장 쉬운 방법은 본보기클라스의 명세부와 실현부를 한개 파일에 포함시

키는것이다. 일부 콤파일러들은 매 파일이 콤파일될 때 본보기함수의 코드가 산생되도록 하는 콤파일시 스위치를 가진다.

9.5 자체평가

- 본보기클라스는 코드의 재리용가능성을 얼마만큼 높이는가?
- 모든 클라스가 본보기화될수 있는가? 왜 그런가?
- 본보기클라스사용에서 어떤 문제가 생기는가?
- 어떤 문서가 본보기클라스로 제공될수 있는가?
- 왜 Stack클라스에서 top메쏘드의 두가지 판본이 있어야 하는가?

9.6 련 습

다음의 클라스를 작성하시오.

- Store

이 클라스는 본보기파라메터로서 다음의것들을 가지는 자료항목들을 기억시킨다.

- 첨수의 형
- 자료항목의 형

메쏘드들은 다음과 같다.

메쏘드	책 임
put	기억기에 [열쇠, 자료]쌍을 추가한다.
get	지적된 열쇠를 사용하여 자료기억기에서 열쇠와 관련된 자료항목을 검색한다.
contains	열쇠가 자료기억기에 존재한다면 참을 돌려 준다.

이 클라스에 대한 명세부는 다음과 같다.

```
#ifndef CLASS_STORE_SPEC
#define CLASS_STORE_SPEC

template <class Index_type, class Item_type>
class Store {
public:
        Store();
        void put( const Index_type key, const Item_type data );
```

```
        Item_type& get( const Index_type key );
        const Item_type& get( const Index_type key );
        bool contains( const Index_type key );
private:
        static const int MAX = 5;
        class Cell
        {
        public:
                Index_type index;                    // 첨수
                Item_type item;                      // 기억된 값
        };
        Cell the_data[ MAX ];                        // 기억되는 곳
        int the_next;                                // 다음첨수
};

#endif
```

이것을 리용하여 다음과 같은 코드부분을 쓸수 있다.

```
Store <std::string, int> parts;

parts.put( "doors", 4 );
parts.put( "sides", 2 );

std::cout << "Number of doors = "
          << parts.get( "doors" ) << "\n";
```

- 개선된 Store클라스

  Store클라스의 구체레를 정의할 때 배렬의 크기를 지적할수 있도록 우의 클라스를 수정하시오.

10 정적변수와 함수

이 장에서는 정적인 기억기클라스에 대하여 서술한다. 이러한 기억기클라스는 그 객체의 수명이 프로그람의 수명과 같다는것을 지정하기 위하여 리용된다.

10.1 정적변수

변수가 선언될 때 그것의 수명은 다음의것들중에서 어느 하나로 된다.

- 프로그람의 수명
  변수가 함수의 밖에서 선언된 경우
- 실행하는 함수의 수명
  변수가 함수의 내부에서(auto로서) 선언된 경우
  기억기클라스 auto에 대해서는 27.2를 보시오.

주의: 클라스에서 선언한 항목의 경우는 후에 설명한다.

그러나 함수가 호출된 다음에 상태정보를 남길 필요가 있을 때가 있다. 물론 이것은 함수밖에서 대역변수를 선언하여 실현할수 있지만 다른 함수가 이 변수를 리용하거나 수정할수도 있다. 변수는 함수안에서 더욱 쓸모 있게 선언될수도 있고 기억기클라스로 주어 질수도 있다. 이것은 실지 함수내부에서만 볼수 있는 대역선언을 생성한다.

실례로 아래의 프로그람에서처럼 함수는 호출될 때마다 새로운 수를 돌려 준다.

```
int unique_number()
{
	static int number = 0;
	return number++;
}
```

주의: 이 선언은 한번만 처리되였다.

10.1.1 클라스에서 정적변수

새로운 클라스구체레가 선언될 때마다 클라스의 모든 자료성원들에 대한 새로운 복사가 진행된다. 많은 경우 이것은 정확히 요구되는것이다. 그러나 클라스의 구체레들이 자료를 공유해야 할 필요가 제기된다. 이것은 대역변수로 이루어 질수 있지만 여기서 위험한것은 프로그람의 다른 부분에서 이 변수를 리용하거나 변경시킬수 있다는것이다.

클라스에서 정적으로 선언된 성원변수는 클라스의 모든 구체레들에 공유된다. 이 정적형태의 성원변수 역시 클라스속성이라고 하는데 매 객체에 대해서가 아니라

전체 클라스의 속성으로 된다.

이러한 처리를 자료성원만을 포함하는 2개의 Ex클라스와 표준자료성원과 정적자료성원을 포함하는 Ex2클라스로 아래에 보여 주었다.

- 프로그람에서 자료성원들만을 포함하는 Ex클라스는 2개의 객체를 선언하는데 리용된다. 프로그람과 결과적인 기억기배치도를 아래에 보여 주었다.

프로그람	기억기배치

Ex one;
Ex two;
 	
int main()
{
    // 코드
}
	Ex의 구체레
	성원함수들

만일 함수가 내부전개(inline)로 선언되였다면 메쏘드를 호출할 때마다 그 코드를 복사한다.

- 프로그람에서 자료성원과 정적자료성원을 포함하는 Ex2클라스는 2개의 객체를 서술하는데 리용된다. 프로그람과 결과적인 기억기배치도를 아래에 보여 주었다.

프로그람	기억기형식

Ex2 one;
Ex2 two;
 	
int main()
{
    // 코드
}
	Ex2의 구체레
	정적자료성원들
	(오직 한번만 복사)
	성원함수들

만일 한 클라스의 성원자료항목이 static로 선언되면 클라스의 구체레가 얼마만큼 만들어 지든 관계없이 오직 정적자료성원의 구체레만이 있을뿐이다.

10.2 검열추적을 가지는 Account클라스

처리된 업무의 총수를 기록하는 클라스 Account는 Account의 모든 구체레에 대한 정적성원자료항목을 리용하여 업무의 수를 기록한다.

이 새로운 Account클라스에 대한 책임은 다음과 같다.

메쏘드	책 임
prelude	거래과정을 재설정한다.
audit_trail	거래과정을 인쇄한다.
account_balance	구좌의 잔고를 돌려 주고 그에 대한 업무를 기록한다.
deposit	구좌에 돈을 저금하고 그에 대한 업무를 기록한다.
set_min_balance	초과출금한계를 설정하고 그에 대한 업무를 기록한다.
Withdraw	충분한 자금이 있거나 초과출금이 허용되는 조건에서만 구좌에서 돈을 출금하고 그에 대한 업무를 기록한다.

새 클라스Account의 C++명세부는 다음과 같다.

```
#ifndef ACCOUNT_SPEC
#define ACCOUNT_SPEC

class Account {
public:
        Account();
        static void prelude();                  // the_no_transactions를 초기화한다
        float account_balance() const;          // 잔고를 돌려 준다
        void deposit( const float );            // 구좌에서 출금한다
        float withdraw( const float );          // 구좌에 저금한다
        void set_min_balance( const float );    // 최소잔고를 설정한다
        static void audit_trail( ostream& );    // 검열추적을 인쇄한다
private:
        float the_balance;                      // 미결제잔고
        float the_min_balance;                  // 최소잔고
        static int the_no_transactions;         // 검열추적을 위하여
};

#endif
```

이 클라스에 2개의 메쏘드인 정적변수 the_no_transaction을 0으로 초기화하는 prelude와 모든 구좌에서 진행된 업무의 총수를 인쇄하는 audit_trail검열추적이 새롭게 추가되였다.

이 함수들은 둘 다 Account구체레의 참조가 없이도 호출될수 있도록 정적으로

선언되였다.

주의: 정적성원함수는 오직 클라스의 정적성원들에만 접근할수 있다. 그러나 파라메터로 넘겨 진 클라스의 구체례에 접근될 때에는 비정적성원들도 접근할수 있다.

정적성원함수는 실현되는 클라스에 메쏘드이름을 붙여 호출된다.

```
Account::prelude();                    // 클라스이름::정적성원함수
```

Account클라스의 실현부는 다음과 같이 시작한다.

```
#ifndef ACCOUNT_IMP
#define ACCOUNT_IMP
#include "Account.h"

int Account::the_no_transactions;

Account::Account()
{
        the_balance          = 0.00;          // 열린 잔고
        the_min_balance      = 0.00;          // 초과출금이 없음
}

void Account::prelude()
{
        the_no_transactions = 0;
}
```

정적변수 the_no_transactions의 명세부는 그 어떤 기억기도 할당하지 않으며 그것은 명시적으로 할당되여야 한다. 기억기할당은 이 클라스의 실현부코드에서 진행된다. 기억기를 이와 같이 명시적으로 선언하는것은 명세부가 분할단위로 콤파일되는 경우 프로그람안에 여러번 포함될수 있기때문이다.

모든 구좌에서 지금까지 진행된 업무수를 인쇄하는 메쏘드 audit_trail의 실현부는 다음과 같다.

```
void Account::audit_trail( ostream& ostr )
{
        ostr << "The total number of transactions was ";
        ostr << the_no_transactions;
        ostr << "\n";
}
```

이 클라스의 나머지실현부는 the_no_transactions가 유효한 매 업무에 대하여 증가된다는것을 제외하고는 이전의 실현부를 그대로 쓴다. 실례로 업무 deposit는 다음과 같이 된다.

```
float Account::withdraw( const float money )
{
	the_no_transactions++;
	if ( money <= the_balance+the_min_balance && money > 0.00 )
	{
		the_balance = the_balance - money;
		return money;
	} else {
		return 0.00;
	}
}
#endif
```

우의 성원함수는 다음과 같이 리용될수 있다.

```
int main()
{
	std::cout << std::setiosflags( std::ios::fixed );		//  x.y 형식
	std::cout << std::setiosflags( std::ios::showpoint );	//  0.10
	std::cout << std::setprecision(2);			// 소수점아래 2자리
	Account mike, corinna;
	float obtained;
	Account::prelude();
	mike.deposit(100.00);
	std::cout << "Mike's  account = " << mike.account_balance() << "\n";
	obtained = mike.withdraw(20.00);
	std::cout << "Mike's  account = "<< mike.account_balance() << "\n";
	mike.deposit(50.00);
	std::cout << "Mike's  account = "<< mike.account_balance() << "\n";
	corinna.deposit(200.00);
	std::cout <<"Corinna's account = "<< corinna.account_balance() << "\n";
	Account::audit_trail( std::cout );
	return 0;
}
```

우의 프로그람을 콤파일하고 실행하면 다음과 같은 결과가 나온다.

```
mike Balance = 100.00
mike Balance = 80.00
```

```
mike Balance = 130.00
corinna Balance = 200.00
The total number of transactions was 8
```

주의: 이미 존재하는 클라스로부터 새로운 클라스를 만들었다. 이것은 계승의 개념을 리용하여 코드의 재리용을 촉진시키는 보다 명백한 방법으로 수행할수 있다. 그것은 11장에서 구체적으로 설명한다.

10.3 자체평가

- 클라스에서 정적자료성원과 보통자료성원사이의 차이점은 무엇인가?
- 정적메쏘드를 배치하는데서 무엇을 제한하는가?
- 만일 클라스에서 정적자료성원을 선언한다면 무엇을 달리해야 하는가?
- 메쏘드는 언제 클라스구체례의 참조없이 호출될수 있는가? 왜 그런가를 설명하시오.

10.4 련 습

다음의 함수를 작성하시오.

- 우연수
  이 함수는 호출될 때마다 새로운 우연수를 돌려 준다. 준(pseudo)우연수는 정적unsigned long형의 중간비트들을 제곱하고 100으로 나누기하여 얻는다. 이것은 0~99범위의 준우연수를 줄것이다.

11 계 승

이 장에서는 이미 존재하는 클라스들을 다시 리용하는 방법에 대하여 보기로 한다. 이 방법은 이미 존재하는 클라스들로부터 새로운 클라스를 만들수 있도록 한다. 이 새로운 클라스를 파생클라스라고 한다. 새로운 클라스에 메쏘드와 성원자료항목들을 추가하여 원래클라스의 기능을 확장할수 있다.

11.1 계산서를 인쇄하는 Account클라스

5.3에서 설명한 클라스 Account에는 구좌잔고에 대한 간단한 계산서를 인쇄하는 기능이 없었다. 새로운 클라스 Account_with_statement는 초기의 Account클라스를 수정, 편집하여 창조할수 있다. 그러나 여기에는 몇가지 결함들이 있다.

- 새로운 클라스 Account_with_statement의 작성자는 원래구좌클라스실현부를 가지고 작업해야 한다. 그러면 작성자는 불가피하게 원래클라스세부를 파고 들어야 한다. 이때 우연히 원래클라스의 자료성원들을 변경시켜 원래메쏘드가 호출될 때 일관성을 잃을수 있다.
- 만일 클라스 Account의 실현부코드가 수정되였다면 클라스 Account_ with _statement는 이러한 변경에 맞게 다시 창조되여야 한다.
- 새로운 클라스를 검사할 때 원래클라스인 Account의 모든 메쏘드들도 다시 검사되여야 한다.

Account_with_statement클라스의 책임은 다음과 같다.

메쏘드	책 임
Account△	구좌의 초기상태를 설정한다.
account_balance△	구좌의 잔고를 돌려 준다.
deposit△	구좌에 저금한다.
set_min_balance△	초과출금한계를 0.00(초과출금이 없다.)으로 설정한다.
withdraw△	구좌에서 출금한다.
statement	구좌잔고에 대한 간단한 계산서를 표현한 문자렬을 돌려 준다.
Account_with_statement	Account_with_statement구체레의 초기상태를 설정한다.

주의: △표식을 한 메쏘드는 Account클라스에 이미 있는것이다.

클라스 Account_with_statement는 다음의 메쏘드와 자료성원들을 가진다.

메쏘드	Account클라스의 모든 메쏘드외에 statement메쏘드
자료성원	• Account클라스의 모든 자료성원 • the_account_name(구좌의 이름을 보관한다.) • the_statement_no(다음계산서의 번호를 보관한다.)

계승기능은 클라스작성자가 이미 존재하는 클라스의 구성요소들을 모두 계승하고 그외에 새로운 기능들(원래클라스를 더욱 전문화하는)을 추가할수 있게 한다. 이러한 처리과정을 그림 11-1에서 보여 주었다.

원래클라스 Account	**추가적인 요소들**	**계승되는 클라스 Account_with_statement**
Account the_balance the_min_balance deposit withdraw account_balance set_min_balance +	the_account_name the_statement_no statement =	**Account_with_statement** the_balance the_min_balance the_account_name the_statement_no deposit withdraw Account_balance set_min_balance statement

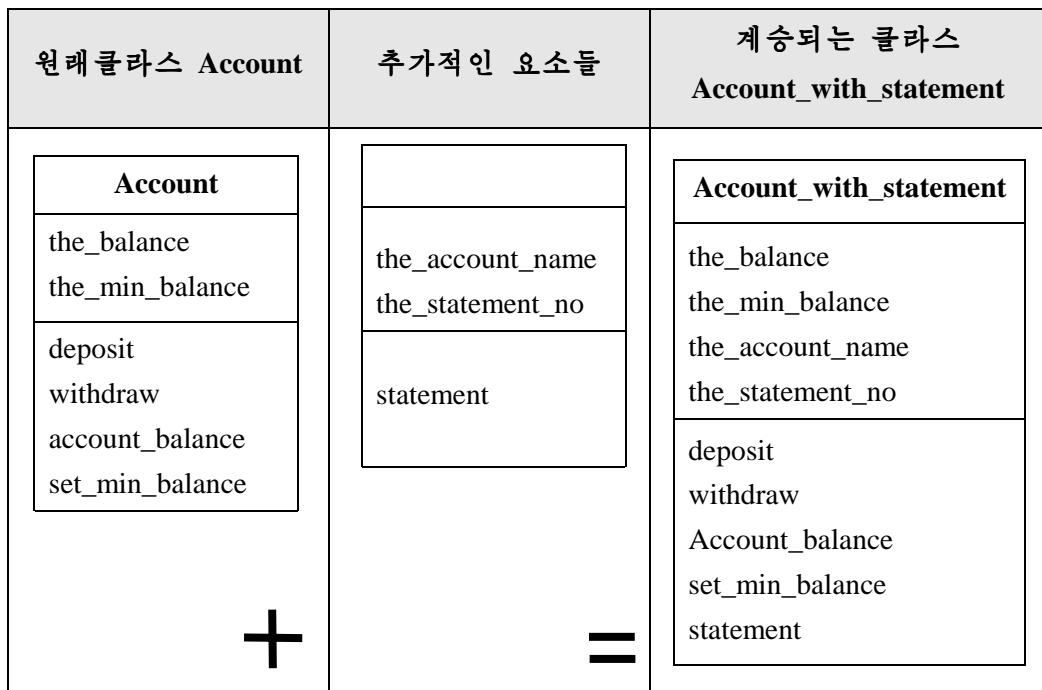



그림11-1. Account_with_statement클라스의 성원들

주의: 이것은 UML수법이 아니라 이미 존재하는 클라스에 요소들을 추가하는 효과를 시각적으로 표시하는것이다.

11.1.1 계승관계를 보여 주는 클라스도표

우의 계승관계를 아래의 그림 11-2에서 UML클라스도표로 보여 주었다.

클라스도표	요소들
Account the_balance the_min_balance deposit withdraw account_balance set_min_balance	클라스 Account는 다음의 구체레변수들과 메쏘드들로 이루어 진다. • 구체레변수들: the_balance, the_min_balances • 메쏘드들: deposit, withdraw, account_balance, set_minbalance
↑	계승을 보여 준다.
Account_with_statement the_account_name the_statement_no statement	클라스 Account_with_statement는 다음의 구체레변수들과 메쏘드들로 이루어 진다. • 클라스 Account의 모든 구체레메쏘드들과 변수들 • 구체레변수 the_account_name과 the_statement_no • 메쏘드 statement

그림 11-2. 클라스Account와 Account_with_statement사이의 관계

11.1.2 용어

계승을 서술하는데서 다음의 용어가 사용된다.

용 어	서 술	실 례
기초클라스	다른 클라스들이 계승하는 클라스	Account
파생클라스	기초클라스로부터 계승되는 클라스	Account_with_statement

11.1.3 Account_with_statement클라스의 명세부

클라스 Account로부터 메쏘드들과 자료성원들을 계승하는 클라스 Account_with_statement의 명세부는 다음과 같다.

```
#ifndef CLASS_ACCOUNT_WITH_STATEMENT_SPEC
#define CLASS_ACCOUNT_WITH_STATEMENT_SPEC
#include <string>
#include <iostream>
#include  "Account.h"

class Account_with_statement : public Account
{
public:
        Account_with_statement( const std::string= "" ); // 구축자
        std::string statement();                          // 계산서를 돌려 준다
private:
```

```
        std::string the_account_name;                    // 구좌의 이름
        int      the_statement_no;                       // 계산서의 번호
};
#endif
```

주의: 명세부의 머리부파일은 클래스 Account의 명세부를 포함한다.

그림 11-3은 이미 정의된 클래스로부터 새로운 클래스가 파생되는 과정을 보여준다.

새로운 클래스 이것의 성원들을 계승한다

class Account_with_statement : public Account

Account성원들에 대한 유효범위변경

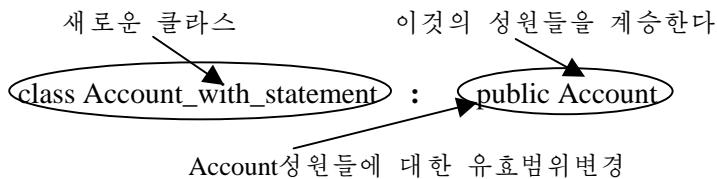



그림 11-3. 클래스 Account로부터 계승한 Account_with_statement

주의: 기초클래스의 성원들에 대한 유효범위변경부(scope modification)에 대해서는 11.5에서 구체적으로 설명한다. 이 경우에 Account_with_statement클라스작성자와 리용자는 Account의 비공개부성원이 아닌 모든 성원들을 볼수 있게 한다.

11.1.4 **Account_with_statement의 실현부**

파생클라스 Account_with_statement의 실현부는 다음과 같다.

```
#ifndef CLASS_ACCOUNT_WITH_STATEMENT_IMP
#define CLASS_ACCOUNT_WITH_STATEMENT_IMP
#include  "Account_with_statement.h"

Account_with_statement::Account_with_statement( const std::string name )
{
        the_account_name = name;
        the_statement_no = 1;
}
```

주의: Account_with_statement의 구축자가 호출되기전에 기초클라스의 Account구축자가 호출된다. 구축자는 형식파라메터를 가지는데 기정값은 빈 문자렬이다. 이 기정값은 클라스의 명세부에서 정의된다.

메쏘드 statement는 구좌잔고에 대한 간단한 계산서(mini-statement)를 문자렬로서 돌려 준다. 파생클라스는 기초클라스의 비공개자료성원인 the_balance를 리용할수 없으므로 구좌잔고를 보내기 위해서는 공개메쏘드 account_balance를 리용한다. 통보 account_balance는 메쏘드 statement가 동작하고 있는 객체에 보내지므로 객체이름은 생략된다. 17.14는 현재메쏘드가 동작하고 있는 객체에 어떻게 직접 접근하는가에 대하여 서술한다.

```
std::string Account_with_statement::statement()
{
   const int MAX_BUF = 100;                          // 본문의 최대크기
   char buf[MAX_BUF];                                // 쓴 본문을 보유한다
   std::ostrstream text( buf, MAX_BUF );             // text는 문자렬흐름이다
   text << std::setiosflags std::ios::fixed );       // 고정소수점 x.y
   text << std::setiosflags( std::ios::showpoint );  // 모든 자리수를 보여 준다
   text << std::setprecision(2);                     // 소수점아래 2자리
   text << "Mini-statement #" <<
                  the_statement_no++ << " for " <<
                  the_account_name << "\n" << "\n" <<
                   "Balance of account is #" <<
                  account_balance() << "\n" << '\0';
         return std::string( text.str() );
}
#endif
```

11.1.5 종합서술

계승은 Account_with_statement클라스를 만들어 내는데 상당한 시간과 노력을 절약한다. 이 클라스에 대한 간단한 시험프로그람을 아래에 보여 주었다.

```
#include <iostream>
//Account_with_statement클라스의 명세부
int main()
{
         Account_with_statement mike("Mike"), corinna("Corinna");
         float obtained;
         mike.deposit(100.00);
         corinna.deposit(250.00);
         mike.deposit(20.00);
         std::cout << mike.statement() << "\n";        // Mike의 계산서
         obtained = mike.withdraw(30.00);
         std::cout << mike.statement() << "\n";        // Mike의 계산서
         std::cout << corinna.statement() << "\n";     // Corinna의 계산서
         return 0;
}
```

주의: 여기서는 mike("Mike")구축자에 값을 넘기는것을 보여 주는데 만일 객체 mike가

Account_with_statement mike;

와 같이 선언되였다면 구축자의 형식파라메터는 기정값을 가진다. 만일 기정값이 정의되지 않았다면 우의 선언은 콤파일시오유로 된다.

우의 프로그람을 콤파일하고 실행하면 다음의 결과가 나온다.

```
Mini-statement #1    for Mike
Balance of account is  £120.00
Mini-statement #2    for Mike
Balance of account is  £90.00
Mini-statement #3    for Corinna
Balance of account is  £250.00
```

11.2 대용관계

클라스 Account_with_statement는 클라스 Account에 대하여 대용(is_a)관계를 가진다. 프로그람작성자는 프로그람에서 클라스 Account의 구체례를 Account_with_statement의 구체례로 대용할수 있으며 프로그람의 효과는 같다. 그러나 한 프로그람에서 클라스 Account_with_statement의 구체례는 항상 클라스 Account의 구체례로 대용할수 없으므로 대칭관계는 아니다.

11.3 저금구좌

은행구좌를 관리하는 클라스 Account가 이미 창조되였으므로 리자산출(interest-bearing)구좌를 어떻게 정의하겠는가에 대하여 보기로 하자. 이 새로운 구좌는 매일 마감에 구좌에 들어 있는 현재잔고에 리자를 지불한다.

리자산출구좌는 보통구좌와 같은데 일리자률을 정의하고 매일리자를 축적하여 그것을 회계주기의 마감에 미결제구좌에 추가하는 기능들을 더 가지고 있다.

아래의 실례에서 기정값으로 정의된 년리자률은 10%이며 기본일리자률은 0.026116%이다. Account에 그러한 기능들을 더 추가한 이 리자산출구좌의 책임들은 다음과 같다.

메쏘드	책 임
interest_accumulate	이때까지 획득한 리자를 축적한다.
interest_credit	구좌에 축적된 리자를 추가하고 축적된 리자를 0.00으로 재설정한다.
end_of_day	매일잔고에 관한 리자를 계산한다. 이 메쏘드는 interest_accumulate메쏘드를 리용하여 축적된 리자를 기록한다.
prelude	일리자률을 설정한다.
set_min_balance	아무런 동작도 하지 않으며 따라서 초과출금을 설정하지 않는다.

이 클라스에 대한 명세부는 다음과 같다.

```
#ifndef CLASS_INTEREST_ACCOUNT_SPEC
#define CLASS_INTEREST_ACCOUNT_SPEC

#include "Account.h"

class Interest_Account : public Account
{
public:
        Interest_Account();
        static void prelude( const float );          // 리자률을 설정한다
        void end_of_day();                           // 리자를 계산한다
        void interest_credit();                      // 구좌에 리자를 추가한다
        void set_min_balance( const float );         // 다중정의(빈 동작)
protected:
        void interest_accumulated_interest;          // 총량
private:
        float the_accumulated_interest;              // 획득한 리자
        static float the_interest_rate;              // 리자률
};

#endif
```

주의: Interest_Account클라스명세부안에 Account명세부를 포함한다.

클라스의 보호부성원들의 역할에 대해서는 11.5에서 론의된다. 본질상 보호부성원은 오직 클라스의 다른 메쏘드와 파생클라스의 메쏘드들에서만 호출할수 있다.

클라스 Interest_Account는 다음의 메쏘드들을 가진다.

Interest_Account클라스에서 정의한것	Account에서 계승된것
Interest_Account interest_accumulate interest_credit end_of_day prelude set_min_balance(주의를 보시오.)	Account account_balance deposit set_min_balance withdraw

주의: Interest_account클라스의 메쏘드 set_min_balance는 기초클라스 Account의 메쏘드 set_min_balance를 다중정의(override)한다.

자료항목들은 다음과 같다.

Interest_Account클라스에서 정의한것	Account에서 계승된것
the_accumulated_interest the_interest_rate	the_balance the_min_balance

우의 Interest_Account클라스에서 볼수 있는바와 같이 이 기구는 상당한 시간과 노력을 절약하며 앞서 정의된 클라스가 새로운 클라스를 만드는데 재리용될수 있도록 한다.

그러나 이것은 이미 정의된 클라스에서 해당조작이 수행될 때에만 재리용되는것이다.

이 새로운 클라스의 실현부는 다음과 같다.

```
#ifndef CLASS_INTEREST-ACCOUNNT_IMP
#define CLASS_INTEREST_ACCOUNT_IMP
#include "Interest_Account.h"

const float RATE = 0.00026116;          // 기정으로 10%리자률을 설정

float Interest_Account::the_interest_rate=RATE;          // 기억기선언

Interest_Account::Interest_Account()
{
        the_accumulated_interest = 0.0;
}
```

prelude메쏘드는 클라스의 정적성원을 초기화하는 기능을 가진다. 만일 클라스의 자료성원이 정적으로 선언되였다면 이 자료항목의 복사는 하나만 존재하며 클라스의 모든 구체례들에 공유된다는것을 상기하시오.

```
void Interest_Account::prelude( const float ir )
{
        the_interest_rate = ir;
}
```

메쏘드 set_min_balance는 기초클라스의 메쏘드를 다중정의한다. 다중정의된 이 함수의 실현부에는 아무것도 없는데 Interest_Account구체례에서 사용자가 최소잔고를 변경시키는것을 막아 준다. 이것은 유효범위해결연산자를 써서 회피할수도 있는데 11.3.3에서 구체적으로 서술한다.

```
void Interest_Account::set_min_balance( const float )
{
        return;
}
```

메쏘드 end_of_day, interest_accumulate와 interest_credit는 구좌에서 리자를 처리하는 기능을 가진다.

매일 작업이 끝나는 때에 메쏘드 end_of_day가 호출되며 메쏘드 interest_accumulate를 리용하여 성원변수 the_accumulated_interest에 리자를 축적한다. 이 축적된 리자는 메쏘드 interest_credit를 사용하여 회계주기의 마감에 손님의 구좌에 저금된다.

```
void Interest_Account::end_of_day()
{
        interest_accumulate(
                account_balance() * the_interest_rate );
}
```

```
void Interest_Account::interest_accumulate( const float ai )
{
        the_accumulated_interest += ai;
}
```

```
void Interest_Account::interest_credit()
{
        deposit( the_accumulated_interest );
        the_accumulated_interest = 0.0;
}

#endif
```

주의: Interest_Account의 매 구체례는 성원변수 the_balance, the_min_balance, the_accumulated _interest를 포함한다.

11.3.1 기초클라스의 구축자호출

Interest_Account형의 객체에 대한 구축자는 구좌에 초기잔고를 설정하기 위하여 Account구축자를 리용한다. 이것은 기초클라스의 구축자가 파라메터를 가지지 않으므로 명시적으로 호출되지 말아야 한다.

```
Interest_Account::Interest_Account()
{
        the_accumulated_interest = 0.0;
}
```

주의: 만일 Account의 구축자가 파라메터를 가진다면 Interest_Account의 구축자는 Account의 구축자에 파라메터가 넘겨 질수 있도록 조금 다른 방법으로 지정된다.

11.3.1.1 구축자와 해체자호출순서

기초클라스구축자가 먼저 호출되고 그다음 파생클라스의 구축자가 호출된다. 객체의 기억기가 해방될 때 파생클라스의 해체자가 먼저 호출되고 그다음 기초클라스의 해체자가 호출된다.

11.3.2 종합서술

다음의 코드는 클라스 Interest_Account의 사용을 보여 준다.

```
// 클라스 Account
// 클라스 Interest_Account
void process();
int main()
{
        const float DAILY_RATE = 0.00026116;                // 년리자률 10%
        Interest_Account::prelude(DAILY_RATE);
        std::cout << std::setiosflags( std::ios::fixed );          // x.y형식
        std::cout << std::setiosflags( std::ios::showpoint );      // 0.10
        std::cout << std::setprecision(2);                         // 소수점아래 2자리
        process();
        return 0;
}
```

다음의 함수 process에서는 클라스 Interest_Account를 시험한다.

```
void process()
{
        Interest_Account mike;
        float obtained;
        mike.deposit(1000.00);
        std::cout << "Account balance=" << mike.account_balance() << "\n";
        obtained = mike.withdraw(200.00);
        std::cout << "Money withdrawn =" << obtained << "\n";
        std::cout << "Account balance =" << mike.account_balance() << "\n";
        mike.deposit(50.00);
        std::cout << "Account balance =" << mike.account_balance() << "\n";
        mike.end_of_day();
        mike.interest_credit();
        std::cout << "Account balance =" << mike.account_balance() << "\n";
}
```

우의 프로그람을 콤파일하고 실행하면 다음의 결과가 나온다.

```
Account balance = 1000.00
Money withdrawn = 200.00
Account balance = 800.00
Account balance = 850.00
Account balance = 850.22
```

주의: 은행구좌는 류점수로서 취급되며 실제량은 소수점아래 두자리까지의 정확도로 취급된다.

11.3.3 다중정의된 성원함수의 호출

파생클라스에서 다중정의된 기초클라스의 성원함수 set_min_balance는 아래와 같이 유효범위해결연산자로써 호출할수 있다.

```
int main()
{
        Interest_Account mike;
        float obtained;

        mike.Account::set_min_balance( -250.00 );
        obtained = mike.withdraw(250.00);

        mike.end_of_day();
        mike.interest_credit();
        std::cout << "Account balance =" << mike.account_balance() <<  "\n";
        return 0;
}
```

우의 프로그람을 콤파일하고 실행하면 다음의 결과가 나온다.

```
Account balance = -250.07
```

주의: 이와 같은 오유는 interest_credit에 의하여 호출되는 메쏘드 deposit가 그의 파라메터가 부인가를 검사하지 않기때문이다. 기초클라스의 메쏘드를 호출하지 못하도록 하기 위한 유일한 방법은 기초클라스를 보호 또는 비공개로 계승하는것이다. 11.5에서 이러한 과정을 서술한다.

11.4 계단식리자률을 가진 저금구좌

기초클라스에서 파생된 클라스는 다른 파생클라스의 기초클라스로 될수도 있다. 실례로 은행은 계단식리자률이 있는 저금구좌를 도입할것을 요구할수 있다. 이러한 형태의 구좌에서 더 많은 돈이 저축되면 더 높은 리자가 구좌에 지불된다. 기초리자률은 Interest_Account의것과 같다.

구좌의 총량	년리자률	기본일리자률(365일을 한해로)
￡10000이하	10%	0.026116%
￡10000에서 ￡24999.99까지	11%	0.028596%
￡25000이상	12%	0.031054%

Interest_Account의 모든 책임외에 이 클라스의 책임은 다음과 같다.

메쏘드	책 임
prelude	일리자률을 설정한다.
end_of_day	하루잔고에 관한 리자를 계산하고 그 구좌에 관하여 이때까지 획득한 리자의 총량을 더한다.

이 클라스의 명세부는 다음과 같다.

```
#ifndef CLASS_SPECIAL_INTEREST_ACCOUNT_SPEC
#define CLASS_SPECIAL_INTEREST_ACCOUNT_SPEC
#include "Interest_Account.h"

class Special_Interest_Account : public Interest_Account
{
public:
        void static prelude( const float, const float, const float );
        void end_of_day();
private:
        static float the_interest_rate1;
        static float the_interest_rate2;
        static float the_interest_rate3;
};

#endif
```

주의: 우의 프로그람에는 명시적으로 정의된 구축자가 없다. 만일 구축자가 제공되지 않는다면 기정구축자가 창조된다. 이 기정구축자는 기초클라스구축자를 호출한다. 우의 경우에는 Interest_Account클라스의 구축자를 호출한다.

메쏘드들에 대한 실현부는 다음과 같다.

```
#ifndef CLASS_SPECIAL_INTEREST_ACCOUNT_IMP
#define CLASS_SPECIAL_INTEREST_ACCOUNT_IMP
#include "Special_Interest.h"

const float R1 = 0.00026116;                    // 10%
const float R2 = 0.00028596;                    // 11%
```

```
const float R3 = 0.00031054;                                    // 12%

float Special_Interest_Account::the_interest_rate1 = R1;
float Special_Interest_Account::the_interest_rate2 = R2;
float Special_Interest_Account::the_interest_rate3 = R3;

void Special_Interest_Account::prelude
            ( const float ir1, const float ir2, const float ir3 )
{
      the_interest_rate1 = ir1;
      the_interest_rate2 = ir2;
      the_interest_rate3 = ir3;
}

void Special_Interest_Account::end_of_day()
{
      float money = account_balance();
      if ( money < 10000 )
            interest_accumulate( account_balance()*the_interest_rate1 );
      else if ( money < 25000 )
            interest_accumulate( account_balance()*the_interest_rate2 );
      else
            interest_accumulate( account_balance()*the_interest_rate3 );
}

#endif
```

메쏘드 end_of_day는 추가적인 기능을 제공하기 위하여 Interest_Account클라스의 end_of_day메쏘드를 다중정의한다. Interest_Account형의 객체가 요구되는 경우에는 클라스 Special_Interest_Account의 구체례를 리용할수 없다. 계승기구는 Interest_Account의 전문화(specialization)를 생성하는데 사용되지만 Interest_Account에 대한 대용(is_a)관계는 아니다.

만일 클라스 Interest_Account의 end_of_day메쏘드를 호출해야 하는 경우 두 함수 end_of_day를 구별하기 위하여 유효범위해결연산자 ::를 Interest_Account::end_of_day()와 같이 리용하여야 한다.

11.4.1 종합서술

다음의 코드는 Special_Interest_Account클라스의 리용을 보여 준다.

```
// 클라스 Special_interest_account

#include <iostream>

void process();

int main()
{
```

```
    const float DAILY_RATE_R1 = 0.00026116;        // 넌리자률 10%
    const float DAILY_RATE_R2 = 0.00028596;        // 넌리자률 11%
    const float DAILY_RATE_R3 = 0.00031054;        // 넌리자률 12%

    Special_Interest_Account::prelude(
        DAILY_RATE_R1, DAILY_RATE_R2, DAILY_RATE_R3 );
    std::cout << std::setiosflags( std::ios::fixed );        // x.y형식
    std::cout << std::setiosflags( std::ios::showpoint );     // 0.10
    std::cout << std::setprecision(2);                        // 소수점아래 2자리

    process();
    return 0;
}
```

```
void process()
{
    Special_Interest_Account mike;
    float obtained;
    std::cout << "Account blance =" << mike.accound_balance() << "\n";
    mike.deposit(20000.00);
    std::cout << "Account balance =" << mike.account_balance() << "\n";
    obtained = mike.withdraw(2000.00);
    std::cout << "Money withdrawn =" << obtained << "\n";
    std::cout << "Account balance =" << mike.account_balance() << "\n";
    mike.deposit(50.00);
    std::cout << "Account balance =" << mike.account_balance() << "\n";
    mike.end_of_day();
    mike.interest_credit();
    std::cout << "Account balance =" << mike.account_balance() << "\n";
}
```

우의 프로그람을 콤파일하고 실행하면 다음의 결과가 나온다.

```
Account balance = 0.00
Account balance = 20000.0
Money withdrawn = 2000.00
Account balance = 18000.00
Account balance = 18050.00
Account balance = 18055.16
```

계승은 응용프로그람에서 코드의 량을 상당히 줄이지만 계승을 효과적으로 리용하기 위하여서는 심중한 계획이 요구된다.

11.5 클라스성원의 보기가능성

어떤 클라스성원들의 외부에서 그리고 그것을 계승할수 있는 다른 클라스에서 볼수 있는가 없는가 하는것은 그 성원들이 클라스안에서 어떻게 선언되였는가에 달려 있다. 사실 보기가능성에는 계층이 존재한다.

```
class Account
{
public:
    // 클라스의뢰자(client of a class)가 볼수 있다.
protected:
    // 계승클라스에서는 볼수 있지만
    // 클라스의뢰자는 볼수 없다.
private:
    // 계승클라스와 클라스의뢰자가 볼수 없다.
}
```

클라스성원들의 보기가능성을 다음의 표에서 보여 준다. 매 경우에 대하여 이 방법으로 서술될수 있는 가능한 항목들을 지적하였다.

클라스성원들의 선언형태	클라스사용자가 클라스 성원을 볼수 있는 가능성	이 방법으로 선언될수 있는 항목들의 실례
public	이 클라스의 유효범위에 있는 모든 항목들이 볼수 있다	클라스사용자가 볼수 있는 함수들
protected	이 클라스의 다른 성원들 그리고 이 클라스를 공개적으로 계승하는 클라스의 성원들이 볼수 있다	클라스사용자가 볼수 있는 함수작성에 리용되는 함수들 그리고 파생클라스에서 리용하는 함수들
private	이 클라스의 다른 성원만이 볼수 있다	파생클라스에서도 리용되지 않는 변수들과 임의의 함수들

주의: 항목들은 보기가능표식을 만날 때까지는 비공개성분으로 간주된다는것이 기정이다.

11.5.1 보기가능성변경자

파생클라스의 의뢰자가 기초클라스성원들을 볼수 있는가 하는것은 파생클라스를 선언할 때 변경될수 있다. 이것은 유효범위변경부 private, protected, public중 어느 하나를 리용하여 해결할수 있다. 아래에 파생클라스 Derived의 의뢰자가 클라스 Base의 기초클라스성원들을 볼수 있는 가능성에 대한 유효범위변경부의 효과를 보여 준다.

	class Derived: △ Base { };와 같이 선언하였을 때 파생클라스 사용자에 대한 유효범위		
Base클라스항목들의 유효범위	△ = public	△=protected	△=private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

그림 11-4에서 public와 private계승에 대하여 보여 주었다. 이 도표에서는 파생클라스에서 기초클라스성원들의 보기가능성을 보여 주었다.

class Derived : **public** Base	**Class** Derived : **private** Base
Base클라스 접근 / **Derived클라스 의뢰자접근** public → public protected → protected private → private	**Base클라스 접근** / **Derived클라스 의뢰자접근** public → private protected → private private → private

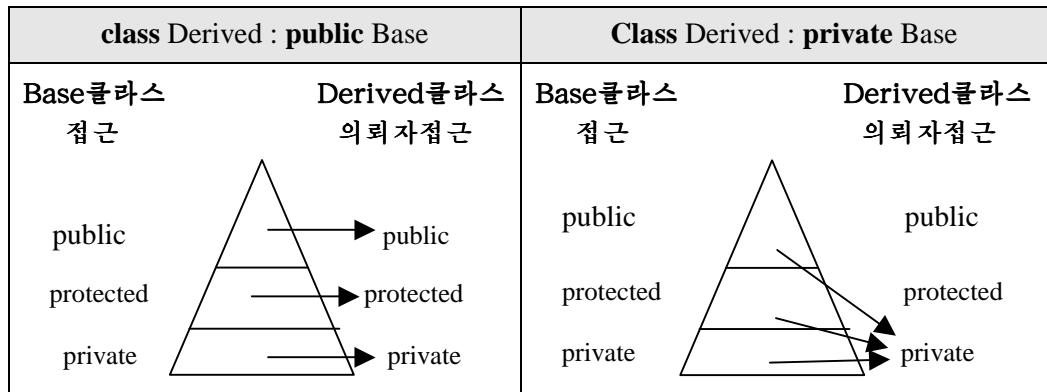



그림 11-4. 기초클라스의 공개부와 비공개부계승

실례로 Base와 Derived클라스명세부를 아래에 보여 주었다.

class Derived : **protected** Base
{
public:
        //파생클라스의 성원들
 	
};
	class Base
	{
	public:
	        int inspect();
	        void mutate();
	protected:
	        void build();
	private:
	        int the_data;
	};

클라스 Base에서 계승된 성원들을 클라스 Derived에서 볼수 있는 가능성은 다음과 같이 된다.

클라스 Base의 성원들	클라스 Derived의 의뢰자가 기초클라스 성원들을 볼수 있는 가능성
inspect, mutate	보호
build	보호
the_data	접근 못함

파생클라스사용자가 기초클라스의 공개성원들을 보지 못하게 하기 위하여 기초클라스를 비공개(privately) 혹은 보호(protectively)로 계승하여야 한다. 기초클라스의 메쏘드에 선택적으로 접근하게 하기 위해서는 이러한 매 메쏘드를 공개(public)다중정의한 판본을 포함한 파생클라스가 요구된다. 파생클라스에서 다중정의한 메쏘드의 실현부는 기초클라스의 성원을 호출한다. 실례로 메쏘드 inspect에는 접근하게 하지만 mutate에는 접근하지 못하도록 하자면 클라스 Derived에 다음과 같은 코드가 요구된다.

```
class Derived : protected Base
{
public:
        int inspect();
        // 파생클라스의 성원들
};

int Derived::inspect()
{
        return Base::inspect();
}
```

주의: 보호계승을 사용하였으므로 메쏘드 inspect와 mutate는 클라스 Derived에서 계승한 클라스에서 볼수 있다.

11.6 구축자와 해체자

앞에서 본바와 같이 클라스의 구축자는 그 클라스의 구체례가 만들어 질 때 지정되였다면 호출할수 있다.

해체자도 역시 그 클라스변수의 구체례가 유효범위를 벗어 나고 기억기가 해제되기전에 호출될수 있도록 지정할수 있다. 클라스의 구체례는 보통 그것이 선언된 블로크가 탈퇴될 때 유효범위를 벗어 난다.

실례로 Room의 책임은 다음과 같다.

메쏘드	책 임
size	방면적을 돌려 준다.
Room	Room의 구체례를 생성한다.
~Room	Room의 구체례를 해제한다.

클라스 Room의 구체례는 프로그람에서 Watts건물의 422번 방을 서술하는데 리용된다. 이 구체례의 이름은 w422이다.

```
watts_building()
{
        Room w422(50);                    // Room의 구체례를 선언

}
```

Room클라스의 구축자는 객체 w422가 선언될 때 호출된다. 이 경우에 구축자는 사무실의 면적을 서술한 위임파라메터를 가진다.

해체자 ~Room은 클라스 Room의 구체례를 포함하는 블로크가 탈퇴될 때 호출된다.

11.7 방을 서술하는 클라스

이 클라스는 건물안의 방에 대하여 서술한다. 클라스구체례는 방의 면적값을 가진다. 메쏘드 size는 방의 면적을 돌려 준다.

구축자는 방의 구체례를 그의 면적크기로 초기화한다. 해체자 ~Room은 방의 구체례가 유효범위밖으로 나가는 사실을 간단히 기록한다.

그러한 클라스의 명세부는 다음과 같다.

```
#ifndef CLASS_ROOM_SPEC
#define CLASS_ROOM_SPEC

class Room {
public:
        Room(const int);                  // 구축자
        ~Room();                          // 해체자
        void size() const;                // 방면적을 표시
private:
        int the_size_sq_metres;           // 면적수
};

#endif
```

이 클라스의 실현부는 다음과 같다.

```
#ifndef CLASS_ROOM_IMP
#define CLASS_ROOM_IMP

Room::Room(const int sq_metres)
{
        the_size_sq_metres = sq_metres;
        std::cout << "Constructor Room      :" <<
                " size in square metres = " << sq_metres << "\n";
}

Room::~Room()
{
        std::cout << "Destructor Room" << "\n";
}

void Room::size() const
{
        std::cout << "Method Room::size()  :" <<
            " size in square metres = " << the_size_sq_metres << "\n";
}

#endif
```

11.7.1 종합서술

우의 클라스 Room은 다음의 함수를 포함하는 프로그람에서 리용된다.

```
void proc_room()
{
        Room w422(50); w422.size();
}
```

우의 프로그람을 콤파일하고 실행하면 다음과 같은 결과가 나온다.

```
Constructor Room       : size in square metres = 50
Method Room::size()    : size in square metres = 50
Destructor Room
```

11.8 사무실을 서술하는 클라스

이 클라스는 클라스 Room에서 파생되며 건물안의 사무실에 대하여 서술한다. 클라스 Office는 사무실면적과 그안에 있을수 있는 직원들의 수로 초기화된다.

이 클라스의 명세부는 다음과 같다.

```
#ifndef CLASS_OFFICE_SPEC
#define CLASS_OFFICE_SPEC
#include "Room.h"
class Office : public Room {
public:
        Office( const int, const int );          // 구축자
        ~Office();                               // 해체자
        void staff() const;                      // 직원수를 표시
private:
        int the_no_staff;                        // 직원수
};
#endif
```

이 클라스의 실현부는 다음과 같다.

```
#ifndef CLASS_OFFICE_IMP
#define CLASS_OFFICE_IMP
Office::Office( const int size, const int no_staff ) : Room(size){
        the_no_staff = no_staff;
        std::cout << "Constructor Office      :" <<
                    "number of staff = " << the_no_staff << "\n"
}
Office::~Office(){
        std::cout << "Destructor Office" << "\n";
}
void Office::staff() const {
        std::cout << "Method Office::staff() : " <<
                    "number of staff = " << the_no_staff << "\n";
}
#endif
```

방의 면적은 Room클라스구축자에 파라메터로 지정되여야 하므로 이 경우에 Office구축자는 Room의 구축자를 암시적으로는 호출할수 없다. 이것은 그림 11-5에서 보여 준것처럼 명시적으로 호출된다.

```
파라메터를 가지는 Office구축자
Office( const int size, const int no_staff ) :    Room(size) {
}                                  처음으로 Room구축자를 호출한다
```

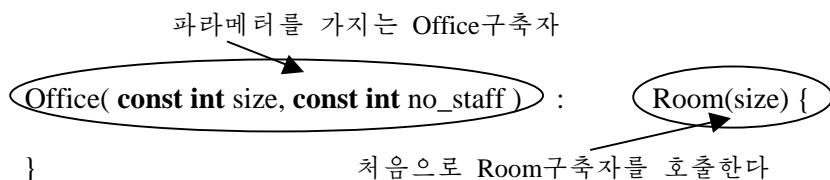



그림 11-5. 기초클라스에서 구축자의 명시적호출

11.8.1 종합서술

우의 클라스 Office는 다음의 함수를 포함하는 프로그람에서 리용된다.

```
void proc_office()
{
        Office w418(24,2); w418.staff(); w418.size();
}
```

함수가 호출되면 다음과 같은 출력을 내보낸다.

```
Constructor Room            : size in square metres = 24
Constructor Office          : number of staff = 2
Method Office::staff()      : number of staff = 2
Method Room::size()         : size in square metres = 24
Destructor Office
Destructor Room
```

주의: 이것은 구축자와 해체자가 기초클라스로부터 파생된 클라스의 구체례에서 호출된 순서를 보여 준다.

11.9 클라스의 자료성원초기화

구축자기구는 객체가 만들어 질 때 객체의 개별적인 자료성원들을 초기화하는데 리용될수 있다. 구축자는 객체가 만들어 질 때 상수자료성원을 초기화할수 있게 한다. 실례로 클라스 Room의 명세부를 다음과 같이 정의할수 있다.

```
#ifndef CLASS_ROOM_SPEC
#define CLASS_ROOM_SPEC

class Room {
public:
        Room(const int );                       // 구축자
        ~Room();                                // 해체자
        void size() const;                      // 방의 면적을 현시
private:
        const int the_size_sq_metres;           // ㎡크기
};

#endif
```

구축자의 실현부는 다음과 같다.

```
Room::Room(const int sq_metres) : the_size_sq_metres(sq_metres)
{
```

```
        std::cout << "Constructor Room     :" <<
                     " size in square metres = " << sq_metres << "\n";
}
```

이것은 자료성원 the_size_sq_metres가 상수로 선언될수 있게 하며 따라서 변이적이지 않도록 한다. 마찬가지로 클라스 Office의 명세부는 다음과 같이 된다.

```
class Office : public Room {
public:
        Office( const int, const int );          // 구축자
        ~Office();                               // 해체자
        void staff() const;                      // 직원수를 표시
private:
        const int the_no_staff;                  // 직원수
};
```

Office클라스에 대한 구축자의 실현부는 다음과 같다.

```
Office::Office( const int size, const int no_staff )
        : Room(size), the_no_staff(no_staff)
{
        std::cout << "Constructor Office     :" <<
                     " number of staff = " << the_no_staff << "\n";
}
```

주의: 이것은 자료성원 the_no_staff의 초기화는 물론 Room의 구축자도 명시적으로 호출한다.

11.10 다중계승

이때까지 새로운 클라스는 하나의 이미 있는 클라스에서 창조되였는데 이미 있는 클라스는 또 다른 클라스에서 창조된것일수도 있다. 어떤 때에는 둘 혹은 그이상의 클라스에 기초한 새로운 클라스를 창조해야 할 필요가 제기된다. 이러한 개념을 다중계승이라고 한다.

실례로 리자산출구좌를 표현한 클라스인 Interest_Account클라스를 생각해 보자. 구좌의 소유자이름은 구좌에 정의되지 않는다. 새로운 클라스인 Named_Account는 아래와 같이 조작할수 있는 Interest_Account클라스와 Person클라스에서 창조될수 있다.

- 클라스 Account의 모든 조작들
- 클라스 Person의 모든 조작들
- 구좌의 소유자에 대한 최소계산서를 돌려 주는 기능

새로운 클라스의 책임에는 클라스 Account와 Person의 모든 책임외에 아래의 내용들이 더 포함된다.

메쏘드 및 구축자	책 임
Named_Account	구좌에 초기값들을 설정
mini_statement	최소계산서의 형태로 구좌의 종합적내용을 문자렬로서 돌려 준다.

이 클라스에 대한 명세부는 다음과 같다.

```
#ifndef CLASS_NAMED_ACCOUNT_SPEC
#define CLASS_NAMED_ACCOUNT_SPEC
#include “Interest_account.h”
#include “Person.h”

#include <string>

class Named_Account : public Interest_Account, public Person
{
public:
        Named_Account( const std::string = “” , const float = 0.0f );
        std::string mini_statement() const;
};

#endif
```

클라스 Named_Account의 메쏘드들은 아래의 표에서 보여 준다.

Interest_Account클라스에서	Person클라스에서	Named_Account클라스
account_balance interest_accumulate interest_credit end_of_day deposit Interest_Account prelude set_min_balance withdraw	address_line lines_in_address name set_address	mini_statement

클라스 Named_Account의 자료성원들은 아래의 표에서 보여 준다.

Interest_Account클라스에서	Person클라스에서	Named_Account클라스
the_interest_rate the_accumulated_interest the_balance the_min_balance	the_details	

Named_Account클라스의 실현부는 다음과 같다.

```
#ifndef CLASS_NAMED_ACCOUNT_IMP
#define CLASS_NAMED_ACCOUNT_IMP
#include <strstream>
Named_Account::Named_Account( const std::string details, const float amount ) :
        Person(details), Interest_Account()
{
        deposit( amount );
}
```

메쏘드 mini_statement는 구좌의 현재잔고를 뒤에 붙인 구좌소유자이름을 문자렬로서 돌려 준다. 이 메쏘드의 실현부는 Person클라스의 메쏘드 name과 Account클라스의 메쏘드 account_balance를 호출한다.

```
std::string Named_Account::mini_statement() const
{
        const int MAX_BUF = 200;                   // 본문의 최대크기
        char buf[MAX_BUF];                         // 문자렬흐름으로서
        std::ostrstream text(buf, MAX_BUF);        // 씌여 진 본문이 들어 있다
        text << “Mini Statement for   : ” << name() << “\n”;
        text << “Balance of account is        : £” << account_balance();
        text << “\n” << ‘\0’;
        return std::string( text.str() );
}
#endif
```

Account, Interest_Account, Named_Account클라스사이관계를 그림 11-6에서 보여 준다.

Account
파생되는쪽
Interest_Account
Person
Named_Account

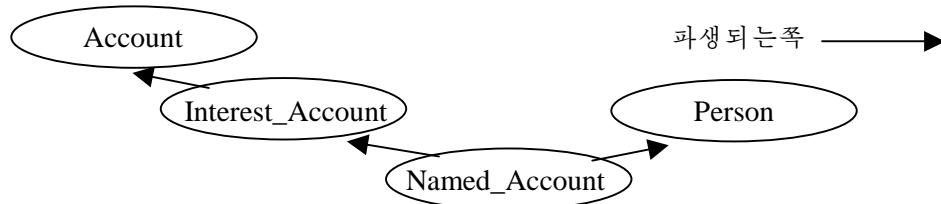



그림 11-6. Named_Account의 클라스계승도

Named_Account형의 객체에서 기초클라스의 구축자를 호출하는 순서를 그림 11-7에서 보여 주었다.

Named_Account::Named_Account(**const** string details, **const** float amount) :

Person(details)　　　　Interest_Account()

첫번째로 호출되는 구축자　　　　두번째로 호출되는 구축자

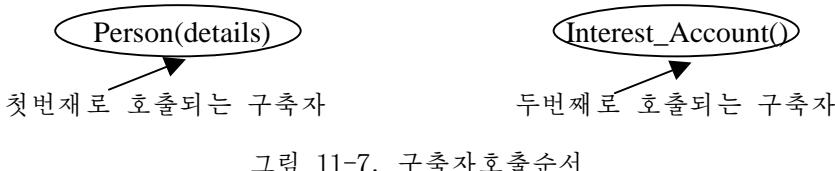



그림 11-7. 구축자호출순서

Named_Account구축자코드본체는 클라스 Account에서 실행되는 deposit메쏘드를 호출하여 구좌에 amount를 저금한다.

주의: 구축자는 왼쪽에서 오른쪽으로 가면서 호출된다.
만일 구축자가 명시적으로 정해 지지 않는다면 기초클라스구축자의 호출순서는 콤파일러가 기초클라스정의를 처리하는 순서이다.

만일 클라스가 심중히 작성되였다면 재리용가능성은 상당히 높아 진다. 그러나 클라스가 요구대로 정확히 작성되지 못하는 경우가 있는데 그때에는 기초클라스를 다시 작성해야 한다.

11.10.1 종합서술

클라스 Named_Account는 프로그람에서 다음과 같이 리용될수 있다.

```
#include <iostream>
void process();                                       // 원형
int main()
{
        const double DAILY_RATE = 0.00026116;         // 넌리자률 10%
        Named_Account::prelude(DAILY_RATE);

        process();
        return 0;
}
```

함수 process는 클라스 Named_Account의 구체례에서 업무처리를 보여 준다.

```
void process()
{
        Named_Account a_n_other(“A N Other/Brighton”);

        std::cout << a_n_other.mini_statement();
        a_n_other.deposit(500);
        std::cout << “Deposit £500”<< “\n”;
```

```
		std::cout << a_n_other.mini_statement();

		a_n_other.end_of_day();
		a_n_other.interest_credit();
		std::cout << "Add interest at end of day"<< "\n";
		std::cout << a_n_other.mini_statement();
}
```

주의: 구좌에 지불될수 있는 리자률변수를 초기화하기 위하여 메쏘드 prelude를 리용한다.

우의 프로그람을 콤파일하고 실행하면 다음의 결과가 나온다.

```
Mini Statement for        : A N Other
Balance of account is     : ￡0.00
Deposit     500
Mini Statement for        : A N Other
Balance of account is     : ￡500.00
Add interest at end of day
Mini Statement for        : A N Other
Balance of account is     : ￡500.13
```

11.11 기초클라스객체에 대한 접근

파생된 객체를 사용할 때 기초객체 혹은 객체들은 독자적인 객체로서 호출된다. 이 기초객체에 대한 접근은 다음의 두가지 방법으로 실현될수 있다.

- 파생객체를 그의 기초클라스들중 어느 하나와 같은 형을 가지는 객체에 값주기하는 방법. 그러면 기초객체만이 값주기된다.
- 형식파라메터형이 기초클라스들중 어느 하나의 형과 같은 함수에 파생객체를 넘기는 방법.

11.11.1 기초클라스객체에 대한 값주기

파생클라스 Named_Account를 리용하여 다음과 같은 코드를 쓸수 있다.

```
Named_Account savings( "M Smith/Brighton/UK" );
Person              mike    = savings;          // Person클라스객체
Interest_Account    ia      = savings;          // Interest_account클라스객체
Account             na      = savings;          // Account클라스객체
```

콤파일러는 savings객체에서 기초객체에 요구되는 일부 요소를 떼낸다. 그러나 다음의것은 콤파일러가 Named_Account구체레에로 Account구체레를 어떻게 변환시켜야 하는지 알수 없기때문에 허용되지 않는다.

```
Named_Account corinna = na                    // 할수 없다
```

그러나 Account형의 형식파라메터를 가지는 Named_Account구축자가 있다면 허용될수 있다.

11.11.2 기초클라스객체에 값대입(암시적값주기)

클라스 Named_Account의 구체레는 마치도 그의 기초클라스들의 구체레인것처럼 리용될수 있다. 실례로 함수 print_label은 Person클라스의 구체레에 대한 주소표식을 인쇄한다.

```
void print_label( std::ostream& ostr, Person& details )
{
        ostr << details.name() << "\n";
        for ( int i=1; i<=details.lines_in_address(); i++ )
        {
                ostr << details.address_line( i ) << "\n";
        }
}
```

함수 print_label은 구좌에서 현재잔고를 인쇄한다.

```
void print_balance( std::ostream& ostr, Account& acc )
{
        ostr << "Balance of account is : £" <<
                        acc.account_balance() << "\n";
}
```

11.11.3 종합서술

이 두 함수는 아래의 코드에서 보는바와 같이 Named_Account형의 실제파라메터를 가지고 호출된다.

```
int main()
{
        Named_Account a_n_other( "A N Other/Brighton/England/BN2 4GJ" );
        a_n_other.deposit(100.00);
        print_label   (std::cout, a_n_other );       // Person클라스로서 처리
        print_balance(std::cout, a_n_other);         // Account클라스로서 처리
        return 0;
}
```

우의 프로그람을 콤파일하고 실행하면 다음과 같은 결과가 나온다.

```
A N Other
Brighton
England
BN2 4GJ
Balance of account is : 100.00
```

11.12 정적맺기

지금까지 본바와 같이 객체에서 메쏘드의 호출과 실행되는 코드사이의 결합은 콤파일시에 평가된다. 이것을 보통 정적맺기라고 한다. 이것은 프로그람작성자가 호출되는 메쏘드의 객체를 미리 안다는것을 의미한다.

실례로 여러 형태의 구좌로 구성된 은행구좌배렬은 관리할수 없다.

16장에서는 여러 형태의 객체배렬을 관리하는 기구에 대하여 본다.

11.13 계승되는 함수

아래의 표는 어떤 함수들이 계승될수 있는가를 보여 준다.

함수의 형태	계승되는가	어떤 함수여야 하는가	기정으로 만들어 지는가
구축자(들)	예	성원함수	예
해체자	아니	성원함수	예
성원함수(들)	예		아니
동료함수(들)	아니		아니

주의: 동료함수에 대하여서는 15.4.2에서 본다.

11.14 같은 기초클라스의 계승

다중계승을 리용하면 클라스가 같은 기초클라스의 다중복사로 만들어 지게 되는 경우가 생긴다. 이러한 현상을 막기 위하여 기초클라스의 이름앞붙이에 virtual예약어를 붙인다. 아래의 실례에서 두 기초클라스 C1과 C2는 다음의 명세부와 실현부를 가진다.

class C1:	**class** C2:

class C1
{
public:
C1();
// C1의 대면부
};
C1::C1() { cout << "C1 "; }
	class C2
	{
	public:
	C2();
	// C2의 대면부
	};
	C2::C2() { cout << "C2 "; }

이 기초클라스들은 두 파생클라스 D1과 D2의 작성에 리용된다.

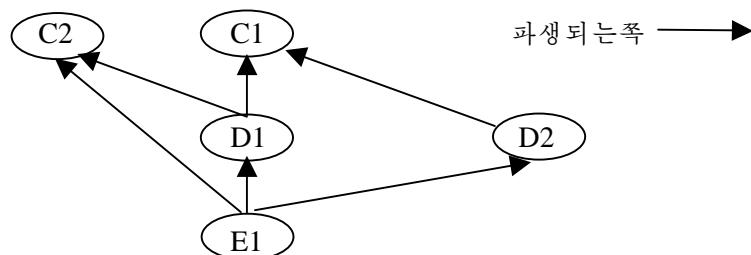
class D1:	**class** D2:

class D1: **virtual public** C1,
　　　　virtual public C2
{
public:
　　D1();
　　// D1의 대면부
};
D1::D1() { cout << “D1 ”; }
	class D2: **virtual public** C1
	{
	public:
	　　D2();
	　　// D2의 대면부
	};
	D2::D2() { cout << “D2 ”; }

주의: 예약어 virtual은 클라스 C1과 C2가 새 클라스를 파생하는데 리용될 때 앞붙이로 쓰인다.

다음으로 클라스 D1, D2, C2는 클라스 C1과 C2를 한번만 복사하는 새 클라스 E1을 파생하는데 리용된다.

```
class E1 : public D1, public D2, virtual public C2
{
public:
        E1();
        // E1클라스의 공개대면부
};
E1::E1() { cout << “E1 ”; }
```

우의 클라스들에 대한 클라스계층도는 다음과 같다.

만일 클라스 E1이 프로그람에서

```
int main()
{
        E1 object;
}
```

와 같이 리용되는 경우 출력결과는 다음과 같다.

```
C1 C2 D1 D2 E1
```

주의: 구축자호출순서는 왼쪽에서 오른쪽으로 가면서 평가된다.

E1 -> D1 D2 C2△ 에서 E1구축자
D1 -> C1 C2 에서 D1구축자
D2 -> C1△ 에서 D2생성자

△이 붙은 클라스는 포함하지 않는다.

virtual예약어를 리용하지 않으면 클라스 C1과 C2가 다중복사된다.
virtual예약어를 쓰지 않은 프로그람을 실행시킨 결과는 다음과 같다.

```
C1 D1 C1 C2 D2 C2 E1
```

주의: 파생클라스 E1에서 클라스 C2앞에 virtual예약어를 생략하였으므로 클라스 C2의 추가복사가 진행되였다.

11.15 자체평가

- 계승의 개념은 쏘프트웨어생산에서 시간을 어떻게 단축할수 있게 하는가?
- 파생클라스의 메쏘드는 기초클라스의 메쏘드와 같은 이름을 가질수 있는가?
- 파생클라스의 성원함수에 의하여 다중정의된 기초클라스의 메쏘드는 파생클라스의 메쏘드로부터 어떻게 호출될수 있는가?
- 정적맺기란 무엇인가?
- 클라스정의에서 공개, 비공개, 보호의 목적은 무엇인가?
- 해체자란 무엇인가?
- 만일 기초클라스가 한개의 구축자를 가지고 새 클라스가 거기에서 파생되였다면 새 클라스는 어떤 구축자를 가지는가?
- 파라메터가 있는 구축자를 가진 기초클라스로부터의 계승을 진행할 때 무엇이 필요되는가?
- 정적메쏘드는 어떤 환경에서 선언하는가?
- 정적성원변수는 어떤 환경에서 선언하는가?
- 만일 C++언어가 다중계승을 가지지 않는다면 클라스 Named_Account를 어떻게 실현할수 있는가?

11.16 련 습

계승을 리용하여 다음의 클라스를 작성하시오.

- Account_with_close
  이 클라스는 Account의 모든 메쏘드들외에 추가로 can_close, close메쏘드를 가진다. 이 메쏘드들의 책임은 다음과 같다.

메쏘드	책 임
can_close	구좌를 닫을수 있는가를 결정한다. 구좌는 초과출금이 없는 경우에 닫을수 있다.
close	구좌의 미결제잔고를 출금한다.

  이것은 구좌에서 초과출금이 진행되지 않는 경우에만 발생한다.

다음의 프로그람을 작성하시오.

- Test
  Account_with_close클라스를 검사하는 프로그람

다음의 클라스를 작성하시오.

- Employee_pay
  종업원의 생활비를 표현하는 클라스 Employee_pay는 다음의 메쏘드를 가진다.

메쏘드	책 임
set_hourly_rate	시간당 값을 설정한다.
add_hours_worked	이때까지 일한 시간수를 축적한다.
pay	그 주에 해당한 지불금을 내여 준다.
reset	일한 시간을 0으로 설정한다.
hours_worked	그 주까지 일한 시간수를 내여 준다.
pay_rate	매 시간당 지불값을 내여 준다.

  총 지불액의 20%를 보험료로 떼낸다.

- Better_employee_pay
  종업원의 생활비을 표현하는 클라스 Better_employee_pay는 클라스 Employee_pay에 다음의 메쏘드를 추가하여 확장하였다.

메쏘드	책 임
set_overtime_pay	초과시간지불값을 설정한다.
normal_pay_hours	초과시간지불값이 적용되지 않는 조건에서 한 주에 일해야 할 시간수를 설정한다.

메쏘드	책 임
pay	그 주에 대한 생활비를 돌려 준다. 이것은 보통생활비값으로 일한 시간과 초과시간비로 일한 시간으로 이루어 진다.

다음의 프로그람을 작성하시오.

- test
  클라스 Better_employee_pay와 Employee_pay에 대한 시험프로그람

다음의 클라스를 작성하시오.

- Employee_pay_with_repayment
  이 클라스는 려행비에 대한 대부금의 일부를 주마다 보상한것을 삭감하여 종업원생활비를 표현하는 클라스이다. 이 클라스는 클라스 Better_employee_pay에 다음의 메쏘드를 추가하여 확장시킨것이다.

메쏘드	책 임
set_deduction	주별 삭감금을 설정
pay	그 주에 대한 지불금을 내여 준다. 이것은 가능하다. 면 종업원대부금에 대한 삭감금까지 계산한것이다.

종업원이 지정된 시간동안 작업하지 못하여 주당 대부금의 보상을 지불할수 없는 경우도 포함한다는것을 생각하시오.

- test
  클라스 Employee_pay_with_repayment에 대한 시험프로그람이다.

- Company
  클라스 Company는 다음의 메쏘드들을 가진다.

메쏘드	책 임
add_hour_worked	매 종업원에 대하여 이때까지 일한 총 시간수를 축적한다.
normal_pay_hours	매 종업원에 대하여 초과시간지불액이 적용되기전에 주에 일하여야 할 시간수를 설정한다.
pay	매 종업원에 대하여 그 주에 대한 지불금을 내여 준다. 이것은 보통생활비값으로 일한 시간과 초과시간비로 일한 시간으로 구성된다.
reset	매 종업원에 대하여 일한 시간을 0으로 설정한다.
set_hourly_rate	매 종업원에 대한 시간당 값을 계산한다.
set_overtime_pay	매 종업원에 대한 초과시간지불액을 설정한다.

s

주의: 이 클라스는 Better_employee_pay클라스의 100개 구체례들의 배렬에 대한 용기클라스이다. 이 클라스구체례에 보내지는 통보문들은 지정된 종업원에 대한 객체를 대표한다.

- Payroll

100명의 종업원을 가지는 작은 기업소에 대한 생활비지불명부를 실현하는 프로그람. 프로그람에서 매 처리단위는 다음과 같은 형태의 한행으로 구성된다.

<동작> <종업원번호> [<파라메터들>]

여기서

<동작>은:

S　그 종업원에 대한 시간당 금액을 설정

O　그 종업원에 대한 초과시간당 금액을 설정

N　그 종업원에 대한 주당 보통생활비로 일한 시간수를 설정. 만일 종업원이 이 시간수이상 일하였다면 그 시간은 초과시간금액으로 지불된다.

A　그 주까지 종업원이 일한 시간을 설정

P　그 주에 종업원이 번 생활비를 인쇄한다. 이것은 총 생활비에서 20%의 보험료를 삭감한것이다.

R　일한 시간수를 0으로 재설정하여 다음주에 대한 자료가 입력되록 한다.

<종업원번호>는:

1부터 100까지의 수이다.

<파라메터들>은:

거래와 관련된 어떤 추가적인 값들이다.

대표적인 처리는 다음과 같다:

S 1 5.00　1번 종업원에 대한 시간당 금액을 5.00파운드로 설정.

O 2 7.50　2번 종업원에 대한 초과시간금액을 시간당 7.50파운드로 설정.

P 2　그 주에 2번 종업원이 번 총 생활비를 인쇄한다(보험료로서 총 생활비의 20%를 삭감한다).

A 3 5　3번 종업원이 그날 일한 시간을 5시간으로 설정한다.

12　4개의 말로 이기는 유희

이 장에서는 객체지향방법을 사용하여 간단한 판유희를 만들어 본다.

12.1　4개의 말

이 간단한 유희는 세로 6칸, 가로 7칸으로 된 판우에서 한다. 매 선수는 그 유희판우의 임의의 칸안에 교대로 서로 다른 색의 말을 하나씩 놓는다. 유희판우에서 수직으로나 수평으로 혹은 대각으로 4개의 말을 먼저 나란히 놓은 선수가 이긴 것으로 된다.

실례로 검은 말과 흰 말을 가진 두 사람이 진행하는 유희판을 보기로 하자.

7번째 수를 쓰고 흰 말을 쓸 차례

		●		●		
	○	●	○	○	●	

9번째 수를 쓰고 흰 말을 쓸 차례

		○				
		●	●	●		
	○	●	○	○	●	

11번째 수를 쓰고 검은말이 승리

		○				
	○	●		●		
	○	●	○	○	●	

평가 :
7번째 수를 쓴 후 흰 말을 4렬에 놓지 못한 것으로 하여 전술상의 오유를 범하였다.
9번째 수를 쓴 후 검은 말이 완전히 이길수 있는 상태에 놓이게 되며 11번째 말을 써서 쉽게 이겼다.

12.1.1　4개의 말로 이기는 유희프로그람

유희의 조종자(유희의 주인)는 교대로 매 선수에게 쓸 수에 대하여 묻는다. 어느 한 선수로부터 쓸 수를 받을 때 유희판은 그 수가 유효한 수인가를 확인한다. 만일 유효한 수라면 그 선수의 말을 유희판에 놓는다. 유희판에는 새로 놓은 말이 표시되며 유희판의 새로운 상태가 평가된다. 이러한 처리는 한 선수가 승리하든가 혹은 유희판에 빈 칸이 없을 때까지 반복된다. 마지막에 수를 쓴 선수는 유희의 결과를 요구한다. 체계와 조종자의 호상작용을 그림 12-1에 보여 준다.

유희판

말 선수1 선수2 말

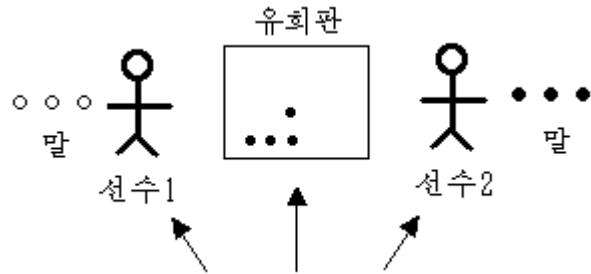



그림 12-1. 체계에서 객체들과 조종자의 호상작용

이 문제에 대한 서술에서 명사는 있을수 있는 객체를 가리키고 동사는 이 객체들에 보내는 가능한 통보를 가리킨다.

굵은(bold)체로 표기된 주요한 **명사**들과 경사(italic)체로 표기된 주요한 ***동사***들을 가지고 이 유희를 상세히 서술하면 다음과 같다.

이 간단한 **유희**는 세로 6 **칸**, 가로 7 **칸**의 **유희판**우에서 ***유희를 한다***. 매 **선수**는 그 **유희판**우의 임의의 **칸**안에 **교대**로 서로 다른 색의 **말**을 하나씩 ***놓는다***. 놓는 **말**은 **칸**에 있는 **말**들이 수직렬이 되도록 서로의 우에 쌓는다. **유희판**우에서 4 개의 말을 수직으로나 수평으로 혹은 대각으로 먼저 만들어 놓는 **선수**는 승리를 ***선포한다***. 유희조종자는 교대로 매 선수에게 쓸 **수**에 대하여 ***묻는다***. 어느 한 **선수**로부터 **쓸 수**를 받을 때 유희판은 그 수가 ***유효한가***를 확인한다. 만일 유효한 수라면 유희판에 그 선수의 **말**을 ***놓는다***. 유희판에는 새로 놓은 말이 ***표시되며*** 유희판의 새로운 상태가 ***평가된다***. 이러한 처리는 한 **선수**가 승리하든가 혹은 **유희판**에 빈 칸이 없을 때까지 반복된다. 마지막에 수를 쓴 **선수**는 **유희**의 결과를 ***선포할***것을 요구한다.

주요객체들과 주요동사들은 다음과 같다

객체(명사)	통지문(동사)
유희판(board) 칸(cell) 말(counter) 선수(player) 유희(game)	선포한다(announce) 묻는다(ask) 표시한다(display) 말을 놓는다(drop) 평가한다(evaluated) 유희를 한다(play) 확인한다(validate)

개별적인 객체들에 다음의 통보문을 보낸다.

유희판(board)	유희판의 상태를 표시한다. 한 칸에 말 한개를 놓는다. 유희판의 현재상태를 평가한다. 신청된 수(놓으려는 말의 위치)를 확인한다.
선수(player)	유희의 결과를 선포한다. 다음수에 대하여 묻는다.
칸(cell)	유희판우의 한 칸안에 말을 한개 놓는다.

말(counter)　　말의 상황을 표시한다.
유희(game)　　유희를 한다.

객체들을 취급하는것보다 클라스들을 취급하는것이 더 편리하다. 실례로 Board 는 객체가 속해 있는 클라스이다. 이 방법을 써서 클라스에 보내는 통보문을 다음의 목록으로 정리해 보자.

클라스	통보문	메쏘드의 책임
Board	display	유희판을 보여 준다.
	drop_in_column	칸에 말을 놓는다.
	evaluate	유희의 현재상태를 평가한다(승리, 비김, 혹은 여전히 경기할수 있음).
	move_ok_for_column	선수가 칸안에 말을 놓을수 있는가를 검사한다.
	reset	유희판을 빈 칸으로 재설정한다.
Player	announce	선수가 경기에서 이겼는가 혹은 비겼는가의 어느 하나를 선포한다.
	counter_is	선수가 쓴 말을 돌려 준다.
	get_move	선수로부터 다음에 쓸 수를 얻는다.
Cell	Clear	칸을 지운다.
	Colour	칸안에 있는 말의 색갈을 돌려 준다.
	Contents	칸안에 있는 말을 돌려 준다.
	Drop	칸안에 말을 놓는다.
Counter	Colour	말의 색을 돌려 준다.
	View	말의 상황을 돌려 준다.
Game	Play	유희를 한다.

주의: 초기통보문(동사)의 일부는 이 목록을 만들 때 더 명확한 이름으로 고치였다.

우에 있는 클라스목록에 있는 메쏘드들을 보면 클라스 Counter 와 Player, Board 에 속한 메쏘드는 명백하게 두 부류로 갈라 진다.

- 입출력을 진행하는 메쏘드 혹은 입출력을 진행하는 객체들과 작용하는 메쏘드.
- 직접적으로나 간접적으로 입출력을 진행하지 않지만 객체의 상태변화를 취급하든가 혹은 상태조사를 진행하는 메쏘드.

클라스 Counter, Player, Board 는 각각 두개의 클라스로 분할되는데 이때 계승이 사용된다.

- 그의 메쏘드가 입출력을 하지 않는 기초클라스.
- 기초클라스로부터 계승되며 입력과 출력을 진행하는 메쏘드를 포함한 파생클라스.

유희 C4 를 만드는데 사용되는 클라스들은 다음과 같다.

클라스	그의 구체례
Board	C4유희의 유희판. 그것은 입출력을 위해 TUI클라스를 써서 외부와 교신한다.
Basic_board	C4유희에 대한 유희판
Player	C4유희의 선수. 그것은 입출력을 위해 TUI클라스를 써서 외부와 교신한다.
Basic_player	C4유희의 선수.
Cell	C4유희판안에 있는 42개 칸들중의 하나.
Counter	C4유희를 하는데 쓰이는 말. 그것은 TUI클라스가 보내는 상황을 돌려 준다.
Basic_counter	C4유희를 하는데 쓰이는 말.
TUI	외부와 대화. 이 클라스를 7.4에서 보여 준다.
Game	C4유희를 진행.

4개의 말로 이기는 유희를 도형대면부를 가지는 유희로 다시 만들수 있도록 클라스들을 여러개로 분할하였다.

12.2 클라스도

4개의 말로 이기는 유희의 클라스도를 그림 12-2에 보여 준다.

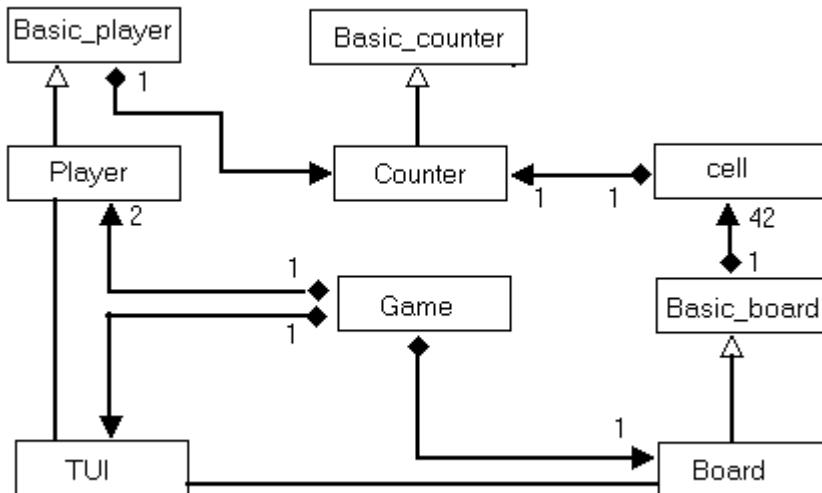



그림 12-2. 4개의 말로 이기는 유희에서 클라스들사이의 관계

주의: 클라스들사이의 선은 UML에 의한 관계를 보여 준다.

12.3 C++클라스에 의한 서술

우에서 고찰한 클라스들에 대한 C++클라스명세부는 다음과 같다.

클라스	C++명세부

Game	**class** Game {
	public:
	  void play ( );
	};

Basic_Counter	**class** Basic_counter {
	public:
	  enum Counter_rep {NONE = ‘ ’ , WHITE = ‘0’,BLACK= ‘x’};
	  Basic_counter (**const** Counter_rep = NONE);
	  Counter_rep colour ( ) **const;** // 말의 색을 돌려 준다
	};

Counter	**class** Counter : **public** Basic_counter {
	public:
	  Counter (**const** counter_rep = NONE);
	  char view ( ) **const;** // 문자로
	};

Player	**class** Player : **public** Basic_player {
	public;
	  Player (Conunt);
	  int get_move (TUI&, **const** bool = **false**) **const;** // 수를 얻는다
	  void announce (TUI&, **const** Board **::** Game_result) **const;** // 결과
	};

Cell	**class** Cell {
	public:
	  Cell ( );
	  void clear ( ); // 칸을 지운다
	  void drop (**const** Counter c); // 칸에 놓는다
	  Basic_counter**::** Counter_rep colour ( ) **const;** // 말의 색
	  const Counter& contents ( ) **const;** // 말
	};

Basic_board	**class** Basic_board {
	public:
	  enum Game_result {DRAW, WIN,PLAYABLE};
	  Basic_board (**const int** rows = DEF_ROWS,
	        const int columns = DEF_COLUMNS);
	  void reset (**const int** rows = DEF_ROWS,
	       const int columns = DEF_COLUMNS);
	  void drop_in_column (**const** int, **const** Counter);
	  bool move_ok_for_column (**const int**) **const;**
	  Game_result evaluate ( ) **const;**
	};

클라스	C++명세부

Board	**class** Board : **public** Basic_board {
	public:
	Board (**const int** rows=DEF_ROWS, **const int** columns=DEF_COLUMNS);
	void Board **::** display (TUI&) **const;**
	}

주의: 우의 C++명세부에서는 공개부(public)성원들만을 보여 주었다.

주클라스 Game의 구체레에 보내는 통보문들에 대한 순서표를 그림 12-3에서 보여 준다.

	Game	Board	Player
유희판을 보여 준다.	display →	▶	
While 유희를 계속할수 있는 조건에서			
선수로부터 쓸 수를 받는다.	get_move →		▶
유효한 수인가를 검사한다.	move_ok_for_column →	▶	
While 유효한 수가 아닌 조건에서.			
유효한 수를 묻는다.	get_move →		▶
유효한 수인가 검사한다.	move_ok_for_column →	▶	
End while			
선수로부터 말의 상태를 얻는다.	counter_is →		▶
유희판에 말을 추가한다.	drop_in_column →	▶	
유희판에 새 상태를 추가한다.	display →	▶	
유희판의 새 상태를 평가한다.	evaluate →	▶	
If 유희를 계속할수 없는 조건에서			
유희의 결과를 선포한다.	announce →		▶
End if			
End while			

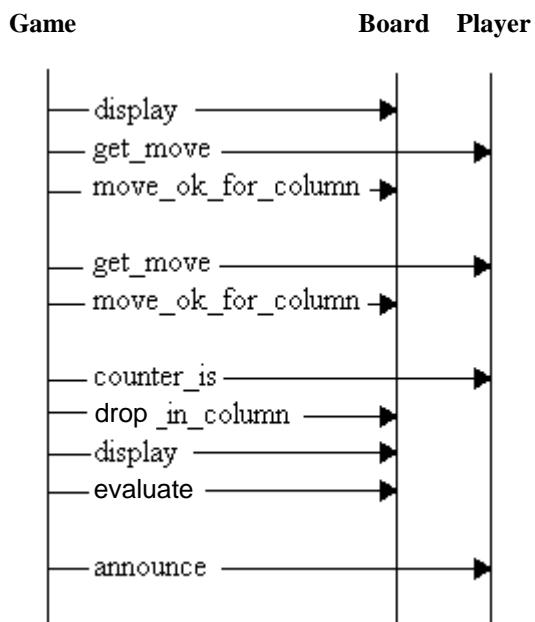



그림 12-3. 유희 C4에 대한 순서표

주의: 순서표는 프로그람안에 있는 객체들에 보내는 통보문들을 보여 준다.

12.4 실 현

앞에서 본 순서표작성작업이 일단 끝나면 그 프로그람에 대한 실제의 코드작성과제는 아주 간단하다. C++의 클라스실현에서 개별적인 클라스들은 여러 프로그람작성자들이 제각기 작성할수 있으나 클라스관계에서는 서로 약속을 해야 한다. 다만 파생클라스의 메쏘드를 실현할 때 기초클라스에 있는 보호부(protected)의 메쏘드만을 사용하는 계승된 클라스는 그 약속에서 제외해도 된다.

12.4.1 Game 클라스

Game클라스에는 두명의 선수가 C4유희를 하는 메쏘드가 있다. 이 클라스의 내용을 모두 서술한 명세부는 다음과 같다.

```
#ifndef CLASS_GAME_SPEC
#define CLASS_GAME_SPEC

class Game
{
public:
   void play();
private:
   TUI     the_screen;                          // 건반
   Board   the_brd;                             // 유희판
   Player  the_ply[2];                          // 선수들
};
#endif
```

메쏘드 play에서는 두명의 선수가 진행하는 C4유희를 쉽게 하도록 클라스 TUI와 Board, Player의 구체례를 사용한다. 이것은 그림 12-3에서 보여 주는 순서도표에 기초하고 있다.

```
#ifndef   CLASS_GAME_IMP
#define   CLASS_GAME_IMP

#include  "Counter.h"                          // 말
#include  "player.h"                           // 선수
#include  "Board.h"                            // 유희판
#include  "Game.h"                             // 유희

void Game :: play ( )
{
    the_ply [0] = player (counter (counter::BLACK ) );      // 검은색말
    the_ply [1] = player (counter (counter::WHITE ) );      // 흰색말

    Board :: Game result game_is = Board :: PLAYABLE;
    int current = 0;                                        // 현재선수

    the_brd.display (the_screen);                           // 유희판을 표시
    while (game_is == Board::PLAYABLE)     // 유희를 계속 할수 있는 동안
    {
        int move = the_ply[current].get_move(the_screen);   // 수를 얻는다
        while (!the_brd.move_ok_for_column(move) )          // 유효이다
        {
            move = the_ply[current].get_move(the_screen, true);
```

```
        }
        Counter plays = the_ply[current].counter_is ( );          // 말을 놓는다
        the_brd.drop_in_column (move, plays);                      // 판에로 추가

        the_brd.display (the_screen);                              // 새판표시
        game_is = the_brd.evaluate ( );
        if (game_is == Board :: PLAYABLE )
          current = (current == 0 ? 1: 0 );                        // 다음선수
        else
          the_ply [current] . announce(the_screen,game_is );   // 결과
      }
}
#endif
```

12.4.2 Basic_counter 클라스

말의 색은 내부에서 Counter_rep로 표현된다. Basic_counter의 구축자는 기정값이 NONE이며 그 기정값은 유희판을 빈 칸으로 만들 때에 사용된다. 또한 유희판우에 말을 놓을 자리가 없을 때에도 사용된다. 이 클라스의 다른 메쏘드는 말의 색을 얻는데 쓰인다. Basic_counter클라스의 명세부는 아래와 같다.

```
#ifndef   CLASS_BASIC_COUNTER_SPEC
#define   CLASS_BASIC_COUNTER_SPEC

class   Basic_counter {
public:
   enum Counter_rep {NONE = ' ' , WHITE == ' 0' , BLAK = ' x'  };
   Basic_counter (const Counter_rep = NONE );
   Counter_rep colour ( ) const;                      // 말의 색을 돌려 준다
private:
   Counter_rep the_colour;                            // 말의 색
};
```

이 클라스의 메쏘드는 효률을 높이기 위해 내부전개(in-line)로 실현된다.

```
inline Basic_counter :: Basic_counter (const Counter_rep colour )
{
   the_colour = colour;
}

inline Basic_counter :: Counter_rep Basic_counter::colour( ) const
{
   return the_colour;
}
#endif
```

Basic_counter클라스의 실현부는 아래와 같다.

```
#ifndef   CLASS_BASIC_COUNTER_IMP
#define   CLASS_BASIC_COUNTER_IMP
#include  "Basic_counter.h"

#endif
```

주의: 클라스의 모든 메쏘드들은 내부전개로 실현되므로 이 파일에는 코드가 들어 있지 않다. 그것은 순전히 호환성을 목적으로 한다.

12.4.3 Counter 클라스

Basic_counter 클라스는 counter로 보다 더 구체화된다. View메쏘드는 TUI 클라스를 사용하여 말의 상태를 표시하는 적합한 출력에 본문자료를 돌려 준다(7.4.4를 보시오).

```
#ifndef   CLASS_COUNTER_SPEC
#define   CLASS_COUNTER_SPEC
#include  "Basic_counter"
#include  <stdexcept>

class Counter : public Basic_counter {

public:
   Counter (const Counter_rep = NONE );
   char view ( ) const;                              //문자로

private:
};
```

주의: 구축자들은 계승되지 않으므로 기초클라스에 있는 구축자에 접근하려 할 때에는 파생클라스의 구축자에서 명백히 호출되도록 써야 한다.

속도를 높이기 위해서 이 클라스의 성원함수도 내부전개로 실현한다.

```
inline Counter :: Counter (const Counter_rep colour )
                  :Basic_counter (colour )
{
}

inline char Counter::view( ) const
{
   switch (colour () )
   {
       case WHITE : return  'o' ;
       case BLACK : return 'x' ;
```

```
        case NONE : return  ‘   ‘;
        default      : throw std :: runtime_error ( “Counter::view” )
                         return  ‘?’ ;                    // 콤파일보존
    }
}
#endif
```

주의: 명령문 throw는 내부오유를 표시하는데 쓰인다. 여기에 대하여서는 14장에서 구체적으로 서술한다.

Counter클라스의 실현부는 다음과 같다.

```
#ifndef   CLASS_COUNTER_IMP
#define   CLASS_COUNTER_IMP
#include  “Counter . h”

#endif
```

주의: 클라스의 모든 메쏘드들이 내부전개로 실현되므로 그 실현부파일에는 코드가 들어 있지 않는다.

12.4.4 Basic_player 클라스

Basic_player 클라스의 명세부는 다음과 같다.

```
#ifndef   CLASS_BASIC_PLAYER_SPEC
#define   CLASS_BASIC_PLAYER_SPEC

#include  “Counter . h”

class Basic_player {
public:
   Basic_player (counter );                         // 선수가 말을 놓는다
   Counter counter_is ( ) const;                    // 선수에게 말을 돌려 준다
private:
   Counter the_players_counter;                    // 말을 놓는다
};
```

counter_is메쏘드는 속도를 높이기 위해 내부전개로 실현된다.

```
inline Counter Basic_player :: counter_is ( ) const
{
   return the_players_counter;
}
#endif
```

Basic_player클라스의 실현부는 다음과 같다.

```
#ifndef   CLASS_BASIC_PLAYER_IMP
#define   CLASS_BASIC_PLAYER_IMP

#include  “Basic_player.h”

Basic_player::Basic_player ( Counter plays_with )
{
   the_players_counter = plays_with;
}

#endif
```

12.4.5 Player 클라스

Basic_player클라스는 Player클라스로 더 구체화되는데 Player클라스는 선수로부터 쓸 수를 입력하고 TUI클라스를 사용하여 유희에 대한 정보를 표시한다.

```
#ifndef   CLASS_PLAYER_SPEC
#define   CLASS_PLAYER_SPEC

#include  "Basic_board.h"
#include  "basic_player.h"
#include  "TUI.h"

class Player : public Basic_player
{
public:
          Player( Counter );
          int get_move(TUI&, const bool=false) const;              // 수를 얻는다
          void announce(TUI&, const Board::Game_result) const;  // 결과
private:
};
#endif
```

Player 클라스의 실현부에서 모든 입출력은 TUI클라스를 통하여 진행된다.

```
#ifndef   CLASS_PLAYER_IMP
#define   CLASS_PLAYER_IMP

#include  <string>
#include  <stdexcept>
#include  "Player.h"

Player :: Player ( Counter plays_with ) : Basic_player ( plays_with )
{
}
```

주의: 구축자는 계승될수 없으므로 기초클라스에 있는 파라메터를 가지는 구축자는 파생클라스의 구축자로부터 호출된다.

get_move메쏘드는 유희를 하는 선수와의 대화를 위해 TUI클라스를 사용한다.

```
int Player :: get_move (TUI& vdu, const bool repeat) const
{
        int move;
        if   ( repeat )                                   // 반복되는 요구
           vdu.message ("invalid last choice");           // 범위
           std::string input = "Move for player ? is ";
           input[16] = counter_is( ).view();              // 선수선택
           vdu.dialogue ( input, move );                  // 수를 요구
        return move-1;
}
```

주의: counter_is().view()표는 counter_is메쏘드로부터 파생된 객체에 통보문 view를 보낸다. 메쏘드 counter_is는 Basic_player클라스에 있다.

메쏘드 announce는 TUI클라스를 사용하여 유희의 결과를 선포한다.

```
void Player :: announce (TUI &vdu, const Board :: Game_result what ) const
{
     switch ( what )
        {
              case Board :: WIN :
              {
                     std :: string result = "Player ? wins";
                     result [7] = counter_is ( ).view( );
                     vdu.message ( result );
                     break;
              }
              case Board :: DRAW :
                     vdu.message ( "The game is a draw" );
                     break;
              case Board::PLAYABLE:
                     vdu.message ( "The game is still playable" );
                     break;
              default :
                     throw std :: runtime_error ("Player::announce");
        }
}
#endif
```

주의: case PLAYABLE은 보통 쓰이지 않는데 유희를 할줄 모르는 사람이 수를 잘못 쓰는 경우에 대처하기 위한 코드이다.

12.4.6 Cell 클라스

유희판에 있는 개별적인 칸에 대한 명세부는 다음과 같다.

```
#ifndef   CLASS_CELL_SPEC
#define   CLASS_CELL_SPEC

#include   "Counter . h"

class Cell {
public:
        Cell ( );
        void clear ( );                                          // 칸지우기
        void drop (const Counter c);                             // 칸안에 놓기
        Basic_counter :: Counter_rep colour ( ) const;           // 말의 색
        const Counter& contents ( ) const;                       // 말
private:
        Counter the_counter;                                     // 말
};
```

클라스 Cell의 구축자와 그의 메쏘드 Clear는 현재칸에 아무 말도 없다는것을 가리키도록 NONE으로 초기화된 counter의 구체례를 사용한다.

이 말의 색은 해당한 성원함수로 검사한다. Cell클라스의 성원함수는 효능을 높이기 위하여 내부전개로 실현한다.

```
inline Cell :: Cell ( )
{
        the_counter = Counter (Basic_counter :: NONE );
}
inline void Cell :: clear ( )
{
        the_counter = Counter ( Basic_counter :: NONE );
}
```

메쏘드 drop는 칸에 검은 말 혹은 흰 말을 놓는다.

```
inline void Cell :: drop (const Counter c )
{
        the_counter = c;
}
```

메쏘드 colour는 칸에 있는 말의 색을 돌려 준다. 돌림값이 NONE인 경우는 칸에 말이 없다는것을 가리킨다.

```
inline Basic_counter :: Counter_rep Cell :: colour ( ) const
{
        return    the_counter.colour ( );
}
```

메쏘드 contents는 클라스 Cell의 메쏘드 counter의 참조를 넘겨 준다. 교갑화를 실현하기 위해 const참조가 넘겨 지므로 counter는 변경될수 없다.

```
inline const Counter& Cell::contents ( ) const
{
        return    the_counter;
}

#endif
```

클라스 Cell의 실현부가 들어 있는 파일은 다음과 같이 정의된다.

```
#ifndef   CLASS_CELL_IMP
#define   CLASS_CELL_IMP
#endif
```

주의: 이 클라스의 모든 성원함수들은 내부전개로 실현되였다.
따라서 클라스 Cell의 실현부파일에는 코드가 들어 있다.

12.4.7 Basic_board클라스

클라스 Basic_board는 이 프로그람에서 제일 복잡한 클라스이다. 그것은 Board의 구체례에 이 유희의 현재상태에 대하여 질문해야 하기때문이다. Basic_board의 실현부에 여러개의 보호부메쏘드들을 더 추가한다.

- 파생클라스에 있는 display메쏘드의 실현부.
- evaluate 메쏘드의 실현부.

```
#ifndef   CLASS_BASIC_BOARD_SPEC
#define   CLASS_BASIC_BOARD_SPEC

#include   "Counter . h"
#include   "Cell . h"

const   int   MAXROW        =9;          // 최대행
const   int   MAXCOLUMN     = 9;         // 최대렬
const   int   DEF_ROWS      = 6;         // 행의 기정값
const   int   DEF_COLUMNS = 7;           // 렬의 기정값
```

```
const   int   STONES_IN_A_WIN_LINE = 4;

class Basic_board {
public:
        enum Game_result { DRAW, WIN, PLAYABLE };
        Basic_board ( const int rows = DEF_ROWS,
                      const int columns = DEF_COLUMNS );
        void reset (const int rows = DEF_ROWS,
                    const int columns = DEF_COLUMNS );
        void drop_in_column( const int, const Counter);
        bool move_ok_for_column( const int ) const;
        Game_result evaluate( ) const;
protected:
        // 분할 및 계승기능
        int max_counters_in_line( ) const;
        int counters_in_dir ( const int, const int, const int,
                              const Basic_counter::Counter_rep ) const;
        // 계승을 위하여
        const Counter& contents ( const int, const int ) const;
        int row_size() const;
        int column_size() const;
private:
        Cell the_grid [MAXROW][MAXCOLUMN];   //경기를 진행한 처음판
        int the_height [MAXCOLUMN];          // 렬에 놓이는 말들의 최대수
        int the_row_size;
        int the_column_size;                 // 경기구역의 크기
        int the_last_col;
        int the_last_row;                    // 다음말을 놓기
        int the_no_empty_cells;              // 빈 칸 없음
};
```

주의: 보호부성원함수는 그 클라스의 내부안에서 사용될뿐아니라 계승되는 클라스에서도 사용할수 있다.

또한 실행속도를 높이기 위해서 아래의 성원함수들을 내부전개로 실현하였다. 이 성원함수들은 파생클라스의 Board display메쏘드의 실현부에 리용된다.

```
inline const Counter& Basic_board::contents ( const int r, const int c ) const
{
        return the_grid[r][c].contents( );
}

inline int Basic_board :: row_size( ) const
{
        return the_row_size;
```

```
}
inline int Basic_board :: column_size( ) const
{
        return the_column_size;
}
#endif
```

주의: 메쏘드 contents는 그 유희판의 칸에 const참조를 돌려 준다.

Board의 실현부는 다음과 같다.

```
#ifndef   CLASS_BASIC_BOARD_IMP
#define   CLASS_BASIC_BOARD_IMP

#include   "Basic_board . h"
#include   <stdexcept>

Basic_board :: Basic_board ( const int rows, const int columns )
{
        reset ( rows, columns );
}
```

메쏘드 reset는 유희판이 빈 칸으로 정의된 상태를 돌려 준다.

```
void Basic_board::reset( const int rows, const int columns )
{
        the_row_size = rows;    the_column_size = columns;
        for ( int c=0; c<the_column_size; c++ )
        {
                for ( int r=0; r<the_row_size; r++ )
                {
                        the_grid[r][c].clear();
                }
                the_height[c] = 0;
        }
        the_last_row            =the_last_col = 0;
        the_no_empty_cells      = rows*columns;
}
```

신청된 수는 메쏘드 move_ok_for_column에 의하여 확인된다.

```
bool Basic_board::move_ok_for_column ( const int column ) const
{
        if   ( column >= 0   &&   column < the_column_size )
        {
```

```
                if   ( the_height [column]   <   the_row_size )
                {
                        return   true;
                }
        }
        return   false;
}
```

메쏘드 drop_in_column은 선택된 렬의 새로운 높이를 기록한 다음 이 렬에 말 C를 추가한다. 이 성원함수는 그 수가 성원함수 move_ok_for_ column에 의하여 확인된 조건에서 사용된다.

```
void Basic_board::drop_in_column ( cont int column, const Counter c )
{
        the_last_col = column; the_last_row = the_height [column];
        the_grid [ the_height [column] ][column].drop (c);
        the_height [column]++;
        the_no_empty_cells--;
}
```

유희판의 현재상태는 성원함수 evaluate에 의하여 결정된다.

```
Basic_board :: Game_result Basic_board :: evaluate ( ) const
{
        return max_counters_in_line ( ) >= STONES_IN_A_WIN_LINE ? WIN :
                ( the_no_empty_cells == 0 ? DRAW : PLAYABLE );
}
```

max_counters_in_line의 방략은 마지막에 말을 놓은 위치로부터 시작하여 색이 같은 모든 말들의 가장 긴 줄을 찾는것이다. 이 함수는 마지막수의 색갈로부터 매개의 둘레방향을 잡아서 말의 개수를 계산하는 순환함수 counter_in_dir를 포함하고 있다. 유희판우의 임의의 위치에 대하여 8개 방향으로 검사한다. 검사하는 선은 말을 놓은 현재의 위치를 통해야 하므로 4개의 선만이 검사에 포함된다. 그림 12-4에 검사해야 할 가능한 방향을 보여 준다.

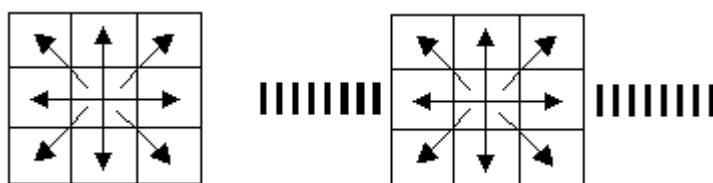



그림 12-4. 어떤 수를 썼을 때 검사해야 할 방향

주의: 매개 선은 서로 반대방향으로 향하는 2개의 선분으로 이루어 져 있다.

성원함수 max_counters_in_line은 제일 길게 련결된 선을 이루는 말의 수를 세기 위하여 서로 반대방향을 따라 계수한 말의 수를 더한다.

```
int Basic_board :: max_counters_in_line( ) const
{
    int max_counters = 0;
    Basic_counter :: Counter_rep counter =
        the_grid[the_last_row][the_last_col].colour ( );
    if   ( counter == Basic_counter :: NONE )   return false;
    for ( int dir = 1;   dir <= 4;   dir ++ )
    {
        int   counters_in_a_line =
              counters_in_dir (dir, the_last_row, the_last_col, counter ) +
              counters_in_dir (dir + 4, the_last_row, the_last_col, counter ) - 1;
        if ( counters_in_a_line > max_counters )
              max_counters = counters_in_a_line;
    }
    return max_counters;
}
```

counters_in_dir는 재귀함수이며 현재의 방향에 말이 더는 없다면 0을 돌려 주며 있으면 다음칸에서 시작하여 현재방향에 있는 말의 수에 1을 더하여 돌려 준다. 이 함수는 매번 말이 놓이는 위치를 보고 유희판앞에 놓이지 않도록 검사를 진행한다.

```
int Basic_board :: counters_in_dir ( const int dir,
                         const int r_cord, const int c_cord,
                         const Basic_counter :: Counter_rep counter ) const
{
        int r = r_cord;   int c = c_cord;
        if   ( ( r >= 0   &&   r < the_row_size ) &&
                 ( c >= 0   &&   c < the_column_size ) &&     // 유희판우에서
               ( the_grid [r][c].colour ( ) == counter ) )       // 선수들의 말
        {
               switch ( dir )
               {
        case   1 :          c++;   break;           //
        case   2 : r ++ ;  c++;   break;           //      8       1       2
        case   3 : r ++ ;         break;           //
        case   4 : r ++ ;  c--;   break;           //      7       .       3
        case   5 :          c--;   break;           //
        case   6 : r -- ;  c--;   break;           //      6       5       4
        case   7 : r -- ;         break;           //
        case   8 : r -- ;  c++;   break;
```

```
            default: throw std::runtime_error ("Board::counters_in_dir" );
            }
            return 1 + counters_in_dir ( dir, r, c, counter );
      } else {
            return 0;
      }
}

#endif
```

주의: 이 함수에 쓰기를 할 때에는 유희판의 경계를 검사한다. case명령문의 default는 내부의 오유를 잡기 위한것이다. 코드검사과정에 이 조건이 발생한다. throw구조에 대하여서는 14장에서 충분히 설명한다.

12.4.8 Board 클라스

클라스 Board는 유희판의 현재상태를 우에서 서술한대로 말단에 인쇄하기 위하여 Basic_board를 확장한다.

```
#ifndef   CLASS_BOARD
#define   CLASS_BOARD

#include   "Basic_board . h"

class Board : public Basic_board {
public:
      Board (const int rows = DEF_ROWS, const int columns = DEF_COLUMNS );
      void Board :: display ( TUI& ) const;
private:
};

#endif
```

클라스 Board의 실현부는 다음과 같다.

```
#ifndef   CLASS_BOARD_IMP
#define   CLASS_BOARD_IMP

#include   <iostream>
#include   <strstream>
#include   "Board . h"

Board :: Board ( const int rows,   const int columns ) :
                  Basic_board( rows, columns )
{
}

void Board :: display ( TUI& vdu ) const
```

```
{
   const int MAX_BUF = 500;                              // 본문구역의 최대크기
   char buf [ MAX_BUF ];                                 // 본문구역
   ostrstream text ( buf,   MAX_BUF );                   // 문자렬흐름

   text   <<   "   ";
   for ( int c = 1;   c <= column_size ( );   c++ )      // 판의 렬수
         text   <<   c   <<   "     ";
   text   <<   "\n";
```

매개 렬에 대한 번호를 출력하고 매칸의 내용을 표시한다.

```
   for ( int r = row_size ( ) - 1 ;   r >= 0;   r -- )    // 유희판의 매행에 대하여
   {
         text << " | ";
         for ( int c = 0; c < column_size ( ); c++ )     // 매렬에 대하여
         {
              text << contents ( r, c ).view ( );        // 칸내용들을 쓰기
              text << " | ";
         }
         text << " \n ";                                 // 행마감
   }
   for ( int c = 1;   c <= column_size ( );   c++ )      // 판의 아래
         text   <<   " ---- ";
   text << " - " << " \n " << " \n" << ' \0 ';

   vdu.message ( text.str ( ) );                         // TUI로 쓰기
}
#endif
```

주의: 클라스 Basic_board에 있는 보호부성원함수는 유희판의 물리적인 상태를 호출하는데 쓰인다.

12.4.9 종합서술

주프로그람은 클라스 Game의 구체레에 통보문 play를 보낸다.

```
#include   "Game . h"
int main ( )
{
         Game ( ).play ( );                              // 유희를 한다
         return 0;
}
```

주의: Game()은 통보문 play를 받는 클라스 Game의 구체레를 넘겨 준다.

대표적인 대화방법은 다음과 같다.

<table>
<tr><th>평가와 수</th><th>수를 쓴후의 유희판</th></tr>
<tr><td>5번째 수를 쓴후 선수 o는 2렬에 자기 말을 놓는다.
선수 o의 수 :2</td><td>1 2 3 4 5 6 7
| | | | | | | |
				x		
	o	x	o	o	x	| ------------------------------</td></tr>
<tr><td>선수x는 다음렬에 자기의 말을 놓는다. 선수 x의 수 :3</td><td>1 2 3 4 5 6 7 |						
		x		x		
	o	x	o	o	x	| ------------------------------</td></tr>
<tr><td>선수 o는 수를 잘못 썼다. 선수 o의 수 :3 선수 x가 말을 놓을 차례 선수 x의 수 :4 이제는 x에 대하여 더는 공격할수 없게 된다.</td><td>1 2 3 4 5 6 7 |						
		o				
		x	x	x		
	o	x	o	o	x	| ------------------------------</td></tr>
<tr><td>선수 o는 3개의 x들이 놓여 있는 행의 어느 한 끝만을 차단할수 있다. 선수 o의 수 :2 선수 x가 승리하였다. 선수 x의 수 :6</td><td>1 2 3 4 5 6 7 |						
		o				
	o	x	x	x	x	
	o	x	o	o	x	| ------------------------------ 선수 x가 이겼다.</td></tr>
</table>

12.5 자체평가

- 클라스 Player에 있는 메쏘드 get_move는 검토자메쏘드이다. 이것은 옳은 생각인가?

- 클라스 Player는 유희판의 구체례를 볼수 있게 하는가? 왜 그런가?
- 도형대면부를 가지고 유희를 할수 있도록 이 유희를 수정하시오. 실례로 유희의 결과가 Web열람프로그람이나 혹은 자기가 알고 있는 다른 도형대면부에 의하여 표시되도록 HTML출력을 만드시오.

12.6 련 습

- 수물리기
  수를 물리고 싶은 사용자의 요구를 들어 주기 위하여 클라스 Game에 있는 메쏘드 play를 수정하시오.

  참고: 이미 쓴 수를 표시하는 Board의 구체례에 대한 배렬을 사용하시오.

계승을 리용하여 다음의 클라스를 작성하시오.

- Board_with_suggested_move
  이 클라스는 Board의 모든 메쏘드와 함께 메쏘드 computers_move를 추가하여 만들수 있다. 이 클라스의 추가적인 책임은 다음과 같다.

메쏘드	책 임
computers_move	콤퓨터가 만드는 수에 대한 렬번호를 돌려 준다.

다음과 같은 프로그람을 작성하시오.

- 콤퓨터유희
  한명의 선수와 콤퓨터가 진행하는 4개의 말로 이기는 유희프로그람.
- 도형식 C4
  한명의 선수와 Web페지의 도형대면부를 사용하는 콤퓨터사이에 진행하는 유희프로그람
- 장기
  두명의 선수가 진행하는 장기프로그람.
- 바둑
  두명의 선수가 진행하는 바둑프로그람.

13 이름공간

이 장에서는 C++의 이름공간에 대하여 서술한다. 이름공간(Namespace)지령은 그 환경에서 이름을 규정대로 취급하도록 하며 이름공간들이 프로그람안에서 많이 쓰이지 않도록 한다.

13.1 이름공간에 대한 소개

C++프로그람에서 사용되는 이름들은 여러개의 서로 다른 이름공간으로 분리될수 있다. 표준서고클라스들은 보통 이름공간 std안에 들어 있다. 실례로 클라스 string을 사용할 때 그의 명세부는 다음과 같다.

```
std::string name = "Corinna";
```

이 선언은 name문자렬이 이름공간 std에 들어 있다는것을 가리킨다. 유효범위 해결연산자(::)는 string이 이름공간 std에 있다는것을 지적하기 위해서 쓰인다. 따라서 이름공간을 쓰면 이름들이 프로그람의 도처에서 되풀이되여 쓰이는것을 방지한다.

실례로 어떤 계의 붉은색표현은 다음과 같은 이름공간 Red로 은페될수 있다.

```
#include   <iostream>
#include   <string>
```

```
namespace   Red                                         // 붉은색
{
        std :: string colour ( )    { return   "red"; }
        std :: string fruit ( )     { return   "strawberries"; }
}
```

한편 계의 푸른색표현은 다음과 같은 이름공간 Blue로 은페될수 있다.

```
namespace   Blue                                        // 푸른색
{
        std :: string colour( )    { return   "blue"; }
        std :: string fruit ( )    { return   "blueberries"; }
}
```

이름공간은 예약어 using에 의하여 프로그람에 반영되는데 그것은 다음과 같다.

```
int main ( )
{
        using namespace Red;
        std :: cout << "My favourite colour is " << colour ( )
                << " and fruit is " << fruit ( ) << "\n";
        return 0;
}
```

실행결과는 다음과 같다.

```
My favourite colour is red and fruit is strawberries
```

만일 우에 서술한 main코드를

```
 int main()
{
        using namespace Blue;
        std :: cout << "My favourite colour is " < colour ( );
                << " and fruit is " << fruit ( ) << "\n";
        return 0;
}
```

로 쓰면 실행결과는 다음과 같다.

```
My favourite colour is blue and fruit is blueberries
```

13.1.1 using지령의 유효범위

예약어 using은 그것이 사용된 단위안에서만 작용한다. 우의 실례에서 예약어 using은 함수안에서 사용되였으므로 그의 유효범위는 그 함수내부이다. 만일 그것이 어떤 단위밖에서 사용되였다면 그의 유효범위는 파일유효범위이다. 파일유효범위라는것은 그 항목이 선언된 위치로부터 파일끝까지 혹은 그 파일이 포함된 파일의 끝까지라는것을 의미한다.

13.2 각이한 이름공간에서 이름의 선택사용

유효범위해결연산자는 개별적인 이름들을 지정된 이름공간들에서 코드렬에 기입하는데 리용된다. 실례로 다음의 코드는 2개의 이름공간 Red와 Blue에서 함수를 사용한다.

```
int main( )
{
```

```
        using Blue :: colour;
        using Red :: fruit;
        std :: cout << "My favourite colour is " << colour ( );
                 << " and fruit is " << fruit( ) << " \n ";
        return 0;
}
```

주의: using의 두 형태
using Blue ::colour ; //이름공간에서 한 항목
using namespace Blue ; //이름공간에서 모든 항목

그것이 콤파일되고 실행되면 다음의 출력을 진행한다.

```
My favourite colour is blue and fruit is strawberries
```

또는 이름공간에서 항목의 호출을 다음과 같이 쓸수 있다.

```
int main( )
{
        std :: cout << "My favourite colour is " << Blue :: colour ( )
                 << " and fruit is " << Red::fruit ( ) << " \n ";
        return 0;
}
```

이것은 이 책에서 보여 준 실례프로그람들에서 이름공간 std에 있는 항목을 호출할 때 사용한 방법이다.

13.3 겹쌓인 이름공간

이름공간은 그자체가 다른 이름공간을 포함해도 된다. 다음의 실례에서 이름공간 Pastel은 이름공간 Yellow와 Pink를 포함한다.

```
namespace Pastel                                        // Pastel색계
{
        namespace Yellow
        {       std :: string colour ( )    { return "light yellow"; }
                std :: string fruit ( )     { return "banana"; }
        }
        namespace Pink
        {
                std::string colour( )       { return "light pink"; }
                std::string fruit( )        { return "papaya"; }
        }
}
```

두개의 이름공간 Yellow와 Red로부터 이름을 지정하는 방법을 아래에 서술한다.

```
int main ( )
{
        using Pastel :: Yellow :: colour;
        using Red :: fruit;
        std :: cout << "My favourite colour is " << colour ( )
                    << " and fruit is " << fruit( ) << " \n ";
        return 0;
}
```

주의: 이름공간 Yellow에 있는 모든 이름을 포함시키기 위하여 using 지령
using Pastel ::Yellow;
을 사용한다.

그것이 실행되면 다음의 결과를 출력한다.

```
My favourite colour is light yellow and fruit is strawberries
```

13.4 별명이름공간

이름공간은 다른 이름으로 별명을 붙일수 있는데 실례로 별명 Favourite는 다음의 선언에 의하여 이름공간 Pastal::Yellow에 대하여 창조된다.

```
namespace Favourite = Pastal :: Yellow;
```

그 별명은 다음의 코드로 쓸수 있다.

```
int main ( )
{
        using namespace Favourite;
        std :: cout << "My favourite colour is " << colour ( )
                    << " and fruit is " << fruit ( ) << " \n ";
        return 0;
}
```

그것이 선택되고 실행되면 다음의 결과를 출력한다.

```
My favourite colour is light yellow and fruit is banana
```

13.5 이름공간의 추가

namespace지령을 사용하여 다른 이름들을 어느 때든지 이름공간에 추가할수 있다. 실례로 이름공간 Blue에 대한 추가는 다음과 같다.

```
namespace Blue                                    // Blue계에 추가
{
        std :: string drink ( )   { return "blue lagoon"; }
}
```

이름공간 Blue는 보통의 방법으로 쓰이는데 실례로 프로그람

```
int main()
{
        using namespace Blue;
        std :: cout << "My favourite colour is " << colour ( ) << " \n ";
        std :: cout << "My favourite fruit   is " << fruit ( ) << " \n ";
        std :: cout << "My favourite drink   is " << drink ( ) << " \n ";
        return 0;
}
```

이 실행되면 다음의 결과를 출력한다.

```
My favourite colour is blue
My favourite colour is blueberries
My favourite drink is blue lagoon
```

13.6 자체평가

- 이름공간 std에는 무엇이 포함되여 있는가?
- namespace 지령을 사용함으로써 프로그람의 관리를 어떻게 개선할수 있는가?
- 프로그람작성자가 서로 다른 여러개의 이름공간으로부터 이름들을 어떻게 선택적으로 사용할수 있는가?

13.7 련 습

- OX유희프로그람의 클라스 Board에 이름공간 Games를 써서 프로그람을 다시 작성하시오.
- C4유희프로그람의 클라스들에 이름공간 Games를 써서 프로그람을 다시 작성하시오.

14 례 외

이 장에서는 C++의 실행시 체계에 의하여 오유와 례외를 처리하는 방법에 대하여 서술한다.

14.1 례 외

례외(exception)는 프로그람에서 특별한 정황으로 인하여 발생되는 사건이다. 실례로 자료구조를 기억기에 할당할 때의 실패가 바로 례외이다. 례외는 잡아 낼수 있으며 그 이례적인 사건은 그것을 취급하는 코드에 의하여 조종할수 있다. 례외는 사실상 프로그람코드가 처리를 중지하고 조종을 호출코드에 돌려 주는 방법으로 실현된다. 호출코드는 간단한 방법으로 이러한 례외를 조종한다. 만일 례외를 포착하지 못하면 례외는 사용자가 작성한 코드로서 처리하거나 혹은 외부체계의 례외조종자가 처리할 때까지 파급되여 간다. 만일 체계의 례외조종자가 례외를 포착하면 그 프로그람은 해당한 오유통보문을 내보내고 실행을 중지한다.

례외는 throw구조에 의해 발생된다. 실례로 사용자가 제공하는 자료를 처리하는 과정에 그것을 기억시킬 기억공간이 불충분하다면 프로그람코드는 처리를 중지해야 한다. 이러한 처리중지를 위해 프로그람작성자는 다음의 명령문을 써서 례외를 일으킨다.

```
throw "Too many items";
```

주의: 이 경우 자료처리코드는 C++문자렬을 내보낸다(throw).

이 코드의 호출자(caller)는 례외를 포착하는 역할을 맡고 있는데 다음과 같은 try블로크에 의해 진행된다.

```
{
        try {
         // 코드부
                throw "Too many items";
        }
        catch ( char exception_mes[] )
        {
                cout << "Fail: " << exception_mes << "\n";
        }
}
```

주의: 내보내진(throw) 항목은 내장자료형이거나 혹은 사용자정의형인 클라스의 어떤 구체례이다. 콤파일러는 내보내진 항목의 림시복사를 만들고 그에 해당한 catch

조종자에 그 림시복사를 보낸다. 이때 그 객체의 안전한 복사를 위해 복사구축자(copy constructor)를 쓸수도 있다. 그런데 try블로크에 생성된(construct) 항목들은 례외가 조종되기전에 해제되며(destruct) 그나마 완전히 생성된 항목들이 해방된다. 그러므로 구축자에서 례외를 발생시키는것은 좋은 방법이 못된다.

우의 실례에서는 C++문자렬이 내보내졌다(throw). C++에서는 문자렬이 문자들의 배렬로서 표현된다. 배렬객체는 그의 첫 문자에 대한 주소로서 표현된다. 따라서 만일 어떤 국부배렬이 내보내진(throw) 객체였다면 내보내진(throw) 실제값은 이 국부배렬의 첫번째 요소에 대한 지적자로 될것이다. 실행시 탄창에 할당된 이 국부배렬의 기억기는 그 함수가 완료된 다음 해방된다. 만일 례외를 포착하는 코드가 그 함수안에 없다면 그때 배렬의 기억기는 catch코드가 실행되기전에 해방된다. 그것은 례외처리부분이 받은 문자렬의 내용에 대하여 정의되지 않은 자료값을 처리하기 때문이다.

그렇지만 문자렬은 대역적인 기억기에서 정적자료항목으로 보유되므로 만일 문자렬상수를 내보내면(throw) 이 문제가 안전하게 된다. 때문에 정적자료항목의 수명은 프로그람의 수명이다.

14.1.1 례외클라스

클라스의 구체례가 정해 진 형태의 오유들을 내보내도록 하기 위하여 다음의 표준적인 클라스들이 제공된다. 이 클라스들은 머리부파일 <stdexcept>에서 정의된다.

- 론리적인 오유를 내보내는 클라스들
  logic_error, domain_error, invalid_argument, length_error, out_of_range
- 실행시 오유를 내보내는 클라스들
  runtime_error, range_error, overflow_error

주의: 이 클라스들에 대한 클라스상수는 례외와 련관된 문자렬파라메터를 가진다. 메쏘드 what는 통보문의 본문을 표현하는 C_str를 돌려 준다.
클라스들은 이름공간 std에서 정의된다.

실례로 실행시 오유를 내보내는 코드는 아래와 같다.

```
int main ( )
{
        try {
                if ( data_values > MAX_DATA_VALUES )
                        throw std :: runtime_error ( "Too many items" );
        }
        catch ( std :: runtime_error& err )
        {
                cout << "Fail: " << err.what ( ) << " \n ";
        }
        return 0;
}
```

례외가 발생하면 다음의 결과를 출력한다.

```
Fail : Too many items
```

14.2 례외의 포착

어떠한 례외를 포착하기 위해서 명령문 catch(…)를 사용한다. 실례로 다음의 코드는 try블로크안에서 호출된 코드에 의하여 내보내진 오유범위를 포착할뿐아니라 발생될수 있는 다른 례외도 포착한다.

```
{
        try {
           // 코드부
        }
        catch ( std :: range_error& err )
        {
             std :: cout << "Fail : " << err.what ( ) << " \n ";
        }
        catch ( ... )
        {
             std :: cout << "Fail : An unexpected exception has occurred" << " \n ";
        }
}
```

주의: catch(…)명령문은 catch명령문의 마지막에 있어야 한다.

14.2.1 클라스 exception에서 파생된 례외의 포착

기초클라스 exception에 있는 catch블로크는 이 클라스에서 파생된 어떠한 례외와도 정합된다.

```
try {
   // 코드부
}
catch ( std :: range_error& err )
{
     std :: cout << "Fail : Range error: " << err.what ( ) << " \n ";
}
catch ( std :: exception& err )
{
     std :: cout << "Fail : Unexpected exception: " << err.what ( ) << " \n ";
}
catch ( ... )
{
```

```
    std :: cout << "Fail: An unexpected exception has occurred" << "\n";
}
```

주의: 여기서도 catch(…)조종자는 표준적인 례외클라스들을 쓰지 않는 례외들을 포착한다.

이 방법으로 예상치 않은 례외를 포착하려면 더 많은 정보가 필요하다.

14.3 전파될수 있는 례외에 대한 서술

현재의 함수에서 례외가 내보내지고 그의 조종자가 거기에 없다면 그 함수를 호출한 코드에서 가능한 조종자에로 례외가 전파된다. 따라서 함수의 정의에서 그 환경밖으로 전파되는 례외의 목록을 서술해야 한다. 실례로 다음의 twice함수에서 례외 overflow_error는 함수 twice의 호출코드로 전파되여야 한다.

```
int twice ( int ) throw ( std :: overflow_error& )
// 최대값설정을 하기 위하여 2의 보수연산을 진행한다
// …
int twice ( int n ) throw ( std :: overflow_error& )
{
        const max_int = ~ ( 1 << (sizeof (int) * 8 - 1 ) );
        if ( n > ( max_int / 2 + 1 ) )
                throw std :: overflow_error ( "Number too big " );
        return   n + n;
}
```

주의: const max_int = ~(1<<(sizeof(int)*8-1));은 2의 보수로서 max_int에 int의 정의옹근수에서 제일 큰 값을 설정한다.
만일 이 함수에 throw의 서술이 없다면 모든 례외는 다른 환경으로 전파된다.

례외 overflow_error는 함수 twice의 호출코드에 전파될수 있다. 그렇지만 례외를 전파시키지 않는 trusting_twice함수는 다음과 같이 정의된다.

```
int trusting_twice (int) throw ( );
int trusting_twice ( int n ) throw ( )
{
        return twice ( n );
}
```

주의: 함수 trusting_twice의 명세부에 있는 throw()는 그 어떤 례외도 그 함수밖으로 전파되지 않는다는것을 정의한다.

14.4 종합서술

알맞는 명세부를 가지고 콤파일되였을 때 앞에서 서술된 함수 trusting_twice와 코드

```
int main()
{
        try {
            std :: cout << "Twice 123    is " << trusting_twice (123) << "\n";
            std :: cout << "Twice 20000 is " << trusting_twice (20000) << " ";
        }
        catch ( std :: overflow_error& err )
        {
            std :: cout << "Fail: " << err.what ( ) << "\n";
        }
        return 0;
}
```

는 다음의 결과를 출력한다.

```
Twice 123        is 246
Abnormal program termination
```

주의: overflow_error의 례외는 함수 trusting_twice밖으로 전파되지 않는다. 그대신에 표준함수 unexpected는 사용자의 말단에 치명적인 오유통보문을 내보내기 위하여 호출된다.

14.5 자체평가

- 례외는 언제 사용하며 언제 사용하지 않는가?
- 프로그람작성자는 내보내진 례외를 어떻게 포착할수 있는가?
- 만일 내보내진 객체가 지적자를 포함한다면 어떤 문제가 생기는가?
- 식 throw 2+3에 의하여 례외가 어떻게 발생되는가, 그것을 포착하시오.
- 프로그람코드로 례외를 포착하지 못하면 어떤 문제가 생기는가?
- 프로그람작성자가 다음의 함수를 어떻게 작성해야 하는가를 설명하시오.
    - ✓ 례외를 전파시키지 않는 함수.
    - ✓ 그 함수가 례외를 전파시키려고 시도한다면 치명적인 오유를 발생시키는 함수.

14.6 련 습

- store

  아래에 서술된 자료항목을 처리하는 클라스를 작성하시오.

메쏘드	책 임
put	[열쇠, 자료]쌍을 기억기에 추가한다.
get	열쇠를 리용하여 자료기억기에서 열쇠와 련관된 자료항목을 찾는다.
contains	열쇠가 자료기억기에 있으면 참을 돌려 준다.

```
# ifndef   CLASS_STORE_SPEC
# define   CLASS_STORE_SPEC

template < class Index_type, class Item_type, const int MAX=5>
class Store {
public:
     Store ( );
     void put(const Index_type key,const Item_type data);
     Item_type& get(const Index_type key);
     const Item_type& get(const Index_type key);
     bool contains(const Index_type key);
private:

};

#endif
```

이 클라스구체례를 사용하는데서 다음의 례외가 발생된다.

예견치 못한 사건	내보내는 례외
항목이 기억되지 않았다.	range_error(“Not there”)
자료기억기가 꽉 찼다.	domain_error(“Full”)

15 연산자의 다중정의

이 장에서는 C++에서 연산자가 새로운 의미를 가지고 다중정의되는 방법을 서술한다. 연산자다중정의는 보통 클라스내부에서 리용되는데 사용자가 언어가 가지고 있는 기능을 확장하여 지정한 방법대로 이 클라스구체레들을 조작할수 있게 한다.

15.1 C++에서 연산자의 정의

money클라스를 정의하는데서 성원함수

```
i_have.add( gift );
```

를 사용하여 더하기를 진행하는것보다도

```
Money i_have, gift;

i_have = i_have + gift;
```

라고 하는것이 더 편리하다.

사용자가 이미 정의된 연산자들을 새로운 기능으로 다중정의해야 할 경우가 추가적으로 생길수 있다.

15.2 클라스 Money

실례로 화페를 취급하는 프로그람에서 기본단위(파운드)와 1/100단위(페니)로 화페량을 취급하도록 새로운 클라스 Money를 정의한다. 더하기연산자(+)는 Money클라스구체레들사이에서 다중정의된다.

클라스 Money의 책임은 다음과 같다.

메쏘드	책 임
print	값을 출력한다.
+	두개의 money객체를 더하여 새로운 객체에 돌려 준다.
++	페니를 하나 증가시킨다.

이 클라스의 C++명세부는 다음과 같다.

```
#ifndef   CLASS_MONEY_SPEC
#define   CLASS_MONEY_SPEC
#include   <iostream>

class Money {
public:
        Money( const long = 0, const int = 0 );          // 구축자
        Money operator + ( const Money );               // 더하기연산자(+)
        Money operator ++( );                           // 앞붙이증가연산자
        Money operator ++(int);                         // 뒤붙이증가연산자
        void print( std::ostream& );                    // 인쇄
private:
        long the_credits;                               // 화폐의 기본단위로
        int the_hundredths;                             // 1/100단위로
};
#endif
```

주의: 증가연산자에는 앞붙이연산자와 뒤붙이연산자라는 두개의 서로 다른 정의가 있다.

Money **operator** ++();	앞붙이증가연산자 ++의 정의
Money **operator** ++(int);	뒤붙이증가연산자 ++의 정의. **주의**: 앞붙이연산자와 구별하기 위하여 가상파라메터가 리용된다.

클라스 Money의 구축자는 다음과 같다.

```
#ifndef   CLASS_MONEY_IMP
#define   CLASS_MONEY_IMP
#include   "Money.h"

Money::Money( const long credits, const int pence )
{
        the_credits = credits;    the_hundredths = pence;
}
```

구축자는 기정값 0으로 정의된 파라메터들을 가지고 있는데 필요한 경우에는 클라스 Money의 구체례가 0이 아니라 미리 정의된 값으로 초기화될수 있다.

성원함수 print는 화폐량에 해당한 형식으로 기억된 값을 선택된 출력흐름에 표시한다.

```
void Money::print( std::ostream& str )
{
   str << "#" << the_credits<<"." << (the_hundredths<10?"0":"") << the_hundredths;
}
```

연산자의 정의(이 경우 +연산자)는 성원함수의 이름대신에 operator +가 지정되는것을 제외하고 성원함수의 정의와 아주 비슷하다.

```
Money Money :: operator + ( const Money p1 )
{
        Money res;
        res.the_hundredths = p1.the_hundredths + the_hundredths;
        res.the_credits     = res.the_hundredths / 100;
        res.the_hundredths = res.the_hundredths % 100;
        res.the_credits      = res.the_credits + p1.the_credits + the_credits;
        return res;
}
```

주의: +는 2항연산자이라고 하여도 두번째 연산수만이 서술되는데 그것은 콤파일러가 아래와 같은 Money구체례를 포함하는 식을 취급하기때문이다.

```
instance_of_money + another_instance_of_money
```

또한 아래와 같은 형식으로도 사용할수 있으나 대체로 쓰지 않는다.

```
instance_of_money.operator + ( another_instance_of_money );
```

++연산자에 대한 코드는 다음과 같다.

```
Money Money :: operator ++ ( )                          // 앞붙이증가연산자
{
        the_hundredths = the_hundredths + 1
        the_credits      = the_credits + ( the_hundredths / 100 );
        the_hundredths = the_hundredths % 100;
        return res;
}

#endif
```

```
Money money :: operator++ (int )                        // 뒤붙이증가연산자
{
   Money res        = *this ;
   the_hundredths = the_hundredths +1;
   the credits       = the_credits+(the_hundredths/100);
   the_hundredths = the_hundredths %100;
   return res;
}

#endif
```

15.2.1 통보를 받는 객체 *this

다중정의된 ++연산자에 대한 코드본체를 작성할 때 그 ++연산자가 연산을 수행하는 클라스의 구체례는 그 결과를 돌려 주어야 한다. 메쏘드중에서 *this는 보내온 통보를 받는 클라스의 구체례이다. 그것을 *this라고 하는 리유에 대해서는 17장에서 설명하는데 거기에는 단항연산자 *에 대한 설명도 들어 있다.

15.3 초기값을 가지는 클라스구체례의 선언

화폐를 취급하는 우의 클라스의 경우에 어떤 초기값 23.45페니를 가지는 클라스 Money의 새로운 구체례를 선언하는것이 편리한 경우도 있다.

이것은 다음과 같이 할수 있다.

```
Money bill ( 23. 45 );
```

사용자가 쓰기 편리하게 생략된 파라메터에는 기정값을 설정할수 있다. revised_bill에 24.00페니의 초기값을 가지도록 선언하기 위해 다음과 같이 쓸수 있다.

```
Money revised_bill ( 24 );
```

또는

```
Money revised_bill = 24;
```

주의: 이러한 형식의 초기화는 같은 구축자를 명시적으로 선언하여 보호할수 있다. 예약어 explicit는 항목이 Money구체례로 암시적으로 변환되는것을 막는다. 이 경우에 24는 클라스 Money의 구체례로 변환된다.

만일 bill이

```
Money bill;
```

로 선언되면 구축자 Money의 두 파라메터는 기정값을 가진다.

주의: 어떤 파라메터에 기정값을 주면 그의 오른쪽옆으로 가면서 있는 다른 파라메터들도 모두 기정값을 주어야 한다.

15.3.1 종합서술

클라스 Money를 사용하여 계산서의 내용을 계산하는 코드는 다음과 같다.

```
#include <iostream>
#include "Money.h"

int main()
{
```

```
        Money ham_pizza( 4, 75 ), extra_cheese = Money( 0, 50 );
        Money tuna_pizza = ham_pizza + Money( 1 );

        std::cout << "A ham pizza costs "; ham_pizza.print( std::cout );
        std::cout << "\n";
        std::cout << "A ham pizza with extra cheese costs ";
        (ham_pizza + extra_cheese).print( std::cout ); cout << "\n";
        std::cout << "A tuna pizza costs "; tuna_pizza.print( std::cout );
        std::cout << "\n";
        return 0;
}
```

주의: 통보 print는 ham_pizza+extra_cheese의 결과로 창조된 객체에 보내진다.

그것이 콤파일되고 실행되면 다음의 결과를 출력한다.

```
A ham pizza costs #4.75
A ham pizza with extra cheese costs #5.25
A tuna pizza costs #5.75
```

15.4 클라스상수

클라스상수는 프로그람에서 상수로서 사용되는 클라스의 구체례이다.
실례로 다음의 코드는 클라스상수 Money(1,0)을 리용하여 화페의 1단위를 표현한다.

```
Money tuna_pizza = ham_pizza + Money( 1, 0 );
```

식 Money(1,0)은 클라스 Money의 림시적인 상수구체례를 발생시키기 위해서 클라스 Money의 구축자를 호출한다. 이 클라스상수의 수명은 실현부에 의존된다.

주의: 콤파일러는 암시적인 클라스상수를 발생시키기 위하여 클라스 Money구축자를 사용할수 있다. 실례로 식

```
Money tuna_pizza=ham_pizza+1;
```

에서 옹근수 1은 클라스상수 Money(1,0)으로 변환된다.

15.4.1 제한

15.2에서 +연산자는 Money형의 왼쪽(LHS-Left Hand Side)에 놓이도록 암시적으로 정의되였다. 대부분의 경우에 이것은 문제로 제기되지 않는다. 이 문제는 friend함수를 리용하여 해결하는데 다음절에서 서술한다.

```
bill = bill + 1;      // 콤파일된다
bill = 1 + bill;      // +의 LHS가 Money형이 아니므로 콤파일되지 않는다
```

주의: 첫번째 경우에는 Money의 구축자가 옹근수를 클라스 Money의 구체레로 변환시키므로 연산된다.

bill=bill+1; => bill=bill.operator+(1).

그러나 두번째 경우에는 콤파일러가 이것을 다음과 같은것으로 취급하므로 콤파일에서 오유가 발생한다.

bill=1+bill; => bill=1. operator+(bill).

15.4.2 동료함수

Money에 대한 클라스를 동료함수(friend)를 사용하여 다음과 같이 정의한다.

```
#ifndef CLASS_MONEY_SPEC
#define CLASS_MONEY_SPEC
#include <iostream>

class Money {
public:
        Money( const long = 0, const int = 0 );
        void print( std::ostream& );                    // 인쇄
        friend Money operator +   ( const Money, const Money );
        friend Money operator ++( Money& );             // 앞붙이증가연산자
        friend Money operator ++( Money&, int );        // 뒤붙이증가연산자
private:
        long the_credits;
        int the_hundredths;
};

#endif
```

동료함수의 주요성질은 다음과 같다.

- 클라스의 성원이 아니지만 마치도 클라스성원인것처럼 그 클라스의 성원들을 호출하게 한다. 즉 클라스구체레안에 있는 비공개부, 보호부의 성원들을 호출할수 있게 한다.
- 2항연산자함수의 LHS와 RHS가 모두 정의되며 이 LHS와 RHS에는 호환변환값주기가 부여된다.
- 그 프로그람에 실제파라메터를 요구되는 형으로 변환하는것이 정의되여 있으면 쌍방향적인 변환이 실시된다. 실례로 이 기능을 수행하기 위한 구축자가 있다.

주의: 동료함수들은 클라스 Money의 성원이 아니기때문에 그 클라스에 부과된 호출제한에 종속되지 않는다. 이러한 《제한 없는 공개》를 제공하기 위하여 동료함수들은 public부에 서술한다.

동료함수의 사용에서 볼수 있는것처럼 다중정의된 연산자인 +와 ++에 대한 파라메터목록에도 LHS가 지정된다.

```
#ifndef CLASS_MONEY_IMP
#define CLASS_MONEY_IMP
Money::Money ( const long credits, const int pence )
{
   the_credits = credits; the_hundredths = pence;
}
void Money::print (std::ostream& str)
{
   str << “#” << the_credits << “ ”
            << (the_hundredths < 10 ? “0” : “ ” )
            << the_hundredths;
}
Money operator + (const Money p1, const Money p2 )
{
   Money res;
   res.the_hundredths = p1.the_hundredths + p2.the_hundredths;
   res.the_credits              =res.the_hundredths /100;
   res.the_hundredths = res.the_hundredths %100;
   res.the_credits                  = res.the_credits + p1.the_credits + p2.the_credits;
   return res;
}
Money operator ++ (Money& p1 )                         // 앞붙이증가연산자
{
   p1.the_hundredths = p1.the_hundredths + 1;
   p1.the_credits + (p1. the_hundredths /100;);
   p1.the_hundredths = p1.the_hundredths % 100;
   return p1;
}

Money operator ++ (Money& p1 )                         // 뒤붙이증가연산자
{
   Money res = p1;
   p1.the_hundredths = p1.the_hundredths + 1;
   p1.the_credits + (p1. the_hundredths /100;);
   p1.the_hundredths = p1.the_hundredths % 100;
   return res;
}

#endif
```

주의: 연산자 +와 ++는 클라스 Money의 성원이 아니므로 그것들을 선언할 때는 유효범위해결연산자를 요구하지 않는다.

15.5 동료함수의 사용

동료함수는 어떤 함수가 그 클라스에 속하지 않으면서도 클라스메쏘드가 가지는 특권을 모두 가질수 있게 한다. 그러므로 동료함수는 다른 클라스의 성원으로 볼수도 있다.

그러나 형검사가 엄격하지 못하고 자료은페가 되지 못한것으로 하여 프로그람작성에서 쉽게 오유를 발생시킬수 있다.

동료함수와 메쏘드

특　징	동료함수	메쏘드
클라스의 모든 성원들에 접근한다	그렇다	그렇다
연산자함수의 모든 연산수들을 서술	그렇다	2항연산자에서 RHS만 지정한다
연산자함수의 인수들에 호환변환값주기가 부여된다	그렇다 (LHS 혹은 RHS)	2항연산자에서 RHS에만 해당된다
클라스의 성원이다	아니다	그렇다

15.5.1 동료클라스

모든 클라스들은 그의 이름을 friend class로 지정하여 다른 클라스의 동료(friend)로 만들수 있다. 실례로 클라스 Account에 어떤 클라스의 모든 성원들을 공개시키려면 클라스명세부에 다음의 행을 포함시켜야 한다.

```
friend class Account;
```

15.5.2 호환변환값주기

클라스 Money를 다음과 같이 수정한다.

```
bill = 1 + Money ( 0, 30 ) + 2;
```

+연산자는 동료함수로 정의되였기때문에 그의 인수들에는 호환변환값주기(assignment compatible conversion)가 부여된다. 1+Money(0,30)에서 1은 구축자 Money(const long = 0. const int = 0)에 의해 Money형으로 변환된다. 다음 다중정의연산자 +는 더하기결과를 넘겨 준다.

주의: 우의 식에서는 Money에 대하여 여러개의 림시적인 클라스상수들이 만들어 진다.

15.5.3 밀기연산자의 다중정의

Money의 구체례에 대한 출력을 C++에서의 일반변수들처럼 진행시키지 위해서 밀기연산자(<<)를 다중정의하는데 이때 클라스 Money에 동료함수를 정의한다. 클라스 Money의 명세부는 다음과 같다.

```
#ifndef CLASS_MONEY_SPEC
#define CLASS_MONEY_SPEC
#include <iostream>

class Money   {
public:
   Money( const long = 0, const int = 0);
   friend   Money operator + ( const Money, const Money );
   friend   Money operator ++ ( Money&);                    // 앞붙이증가연산자
   friend   Money operator ++ ( Money&, int );              // 뒤붙이증가연산자
   friend   std::ostream& operator << ( std::ostream& s, const Money m );
private:
   long the_credits;                                        // 기본단위로
   int     the_hundredths;                                  // 1/100단위로
};
#endif
```

연산자 <<는 다음과 같이 다중정의된다.

```
std::ostream& operator << ( std::ostream& s , const Money m )
{
   s <<  " # " << m.the_credits <<  " . "
      << (m.the_hundredths < 10 ?  "0"  :  " "  )
      << m.the_hundredths;
   return s;
}
```

주의: <<연산자를 다중정의하여 Money형항목들의 내용을 C++의 형식화된 출력으로 인쇄할수 있다.

다중정의된 연산자가 다음과 같은 식으로 사용되도록 연산자함수는 ostream형의 객체에 참조를 보낸다.

```
cout <<  "The value of the item is  " << Money ( 9, 99 ) <<  " \n";
```

파라메터로서 넘어 간 흐름객체는 복사할수 없다.

15.5.4 종합서술

마지막으로 완성된 Money클라스는 다음과 같다.

```
// 마지막으로 정의한 Money클라스

int main ( )
{
   Money ham_pizza ( 4, 75 ), extra_cheese = Money ( 0, 50 );
   Money tuna_pizza = ham_pizza + 1;

   std::cout <<  “A ham pizza costs  “ << han_pizza <<  “ \n”;
   std::cout <<  “A ham pizza with extra cheese costs  “  <<
                     (ham_pizza+extra_cheese) <<  “\n”;
   std::cout <<  “A tuna pizza costs  “ << tuna_pizza <<  “\n”;
   return 0;
}
```

실행하면 다음의 결과가 출력된다.

```
A ham pizza costs #4.75
A ham pizza with extra cheese costs #5.25
A tuna pizza costs #5.75
```

15.5.5 연산자다중정의의 함수적표기법

다중정의된 연산자들은 함수적인 표기법을 사용하여 표현될수 있다. 실례로 클라스 Money에서 다중정의된 연산자 +와 ++는 다음과 같이 정의된다.

```
Money operator + ( const Money );          // +연산자
Money operator ++ ( );                     // 앞붙이증가연산자
Money operator ++ ( int );                 // 뒤붙이증가연산자
```

함수적표기법을 사용하여 프로그람작성자는 다음과 같이 쓸수 있다.

```
Money prefix, postfix, sum, cheese, onion;

prefix.operator++ ();                      // 앞붙이증가연산자
postfix.operator++(1);                     // 뒤붙이증가연산자
sum = cheese.operator+( onion );           // cheese와 onion의 합
```

주의: 일반적으로 이 문법은 사용되지 않는다. 그러나 현재정의에서 숨겨 진 다중정의된 연산자를 사용하는것이 필요되는 경우가 있다. 이 경우에 유효범위변경연산자(scope modification operator)와 결합된 함수적인 표기법은 숨겨 진 함수에 접근될수 있게 한다.

클라스에서 우의 연산들이

```
friend Money operator + ( const Money, const Money );
friend Money operator ++ ( Money& );            // 앞붙이증가연산자
friend Money operator ++ ( Money&, int );       // 뒤붙이증가연산자
```

로 서술되면 함수적인 표기는 다음과 같이 쓸수 있다.

```
Money prefix, postfix, sum, cheese, onion;

prefix.operator++ (prefix);                     // 앞붙이증가연산자
postfix.operator++(postfix, 1);                 // 뒤붙이증가연산자
sum = cheese.operator+(cheese, onion );         // cheese와 onion의 합
```

주의: 다중정의된 연산자 ++에 대하여 앞붙이증가연산자와 뒤붙이증가연산자를 구별하기 위하여 추가파라메터(이 경우에는 1)를 사용한다.

15.6 연산자다중정의에 대한 제한

C++언어에서 쓰이는 연산자만 연산자다중정의에 사용될수 있다. 연산자를 정의할 때 연산자우선권을 지적할수 없으므로 연산자들은 자기들의 본래우선권을 가진다.

다음의 연산자들을 제외한 모든 연산자들(부록 10을 보시오.)은 다중정의된다.

.   .*   ::   ?:

연산자가 다중정의될 때 연산수들중 적어도 한개는

- 클라스형 혹은 클라스형에 대한 참조
- 렬거형 혹은 렬거형에 대한 참조

이여야 한다.

이것은 두개의 int형연산수들사이에 있는 +가 다중정의되지 않게 한다.

15.7 변환연산자

앞의 실례에서 값주기연산자를 사용하여 클라스의 구체례를 한 기억기에서 다른 기억기에로 복사하였다. 이 코드는 새로운 기억위치에 클라스의 구체례에 의하여 표시되는 기억기를 간단히 복사하였다.

주의: 값주기연산자를 다중정의하는 방법은 23장에서 보여 준다. 거기서는 클라스구체례에 대한 내용들을 복사하는 또 다른 방법들을 보여 준다.

클라스 Money를 사용하여 다음의 코드를 쓸수 있다.

```
Money mine, yours;
mine = Money ( 10, 50 )
yours = mine;
```

파운드와 프랑에 대한 클라스들이 있다면 다음과 같이 쓸수 있다.

```
Pounds to_spend ( 10, 50 );                    // 영국화페
Francs holiday_money;                          // 스위스화페
holiday_money = to_spend;
std::cout << "Holiday money = " << holiday_money << "\n";
```

to_spend에 의해 넘겨 지는 값은 요구값에 따른다. 우의 경우에서 to_spend의 요구값은 스위스프랑값이다. 마찬가지로 화페들사이에 형변환을 리용하는것이 편리하다. 실례로 스위스프랑을 딸라로 변환하기 위해 프로그람작성자는 다음과 같이 쓸수 있다.

```
std:: cout << "Holiday money = " << (Francs) to_spend << " \n";
```

이것을 실현하기 위하여 변환연산자가 클라스 Pounds에서 정의되는데 이때 각이한 형의 객체들을 서로 변환하는 코드를 포함한다. 우의 화페변환실례에서 연산자는 두 화페들사이의 교환비률을 사용한다.

15.7.1 변환연산자의 명세부와 실현부

Pounds형객체를 Francs형객체로 변환하는 클라스 Pounds의 변환연산자는 다음과 같이 선언된다.

```
class Pounds
{
public:
   operator Francs( ) const ;                  // 변환연산자 Francs
};
```

그리고 실현부는 다음과 같다.

```
Pounds::operator Francs() const
{
   long francs; int centime;
     // 파운드를 프랑으로, 페니를 쌍팀으로 변환
   return Francs ( francs, centime );
}
```

주의: 이것은 클라스 Francs에 구축자가 있어 프랑과 쌍팀(1/100프랑)으로 이루어 진 Francs의 구체레를 생성한다고 가정한다.

15.8 기초클라스로서의 Money클라스

딸라, 스위스프랑, 파운드를 취급하는 프로그람작성에서는 그림 15-1과 같은 계승구조가 리용된다.

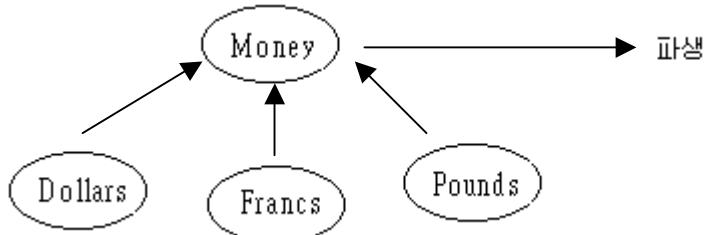



그림 15-1. 딸라, 스위스프랑, 파운드에 대한 클라스계층구조

그러나 앞에서 정의한 클라스 Money는 다음의 리유들로 하여 사용될수 없다.

- 동료함수들은 계승되지 않는다. 이것은 +가 Pounds형객체사이에 사용된다면 Money클라스에서 동료함수를 사용하기전에 연산수들은 Money형으로 변환되여야 한다는것을 의미한다. 그 결과는 Pounds형으로 다시 변환되여야 한다. 이렇게 하면 불필요한 변환들이 많이 생긴다.
- 계승함수에서 돌림값형은 계승하는 클라스의 형으로 바꾸어 지지 않는다. 실례로 Money클라스의 함수

  Money operator + (Money)

  는

  Money operator+(pounds).

  로서 pounds에 계승된다.

따라서 새로운 클라스 Money를 작성하여야 한다. 이 새로운 클라스 Money의 명세부는 다음과 같다.

```
#ifndef   CLASS_MONEY_SPEC
#define   CLASS_MONEY_SPEC

class Money {
public:
        explicit Money( const long = 0L, const int = 0 );
        static void add( Money&, Money, Money );
        friend std::ostream& operator <<( std::ostream&, const Money );
        long units( ) const;                 // 화페의 기본단위를 돌려 준다
        int hundredths( ) const;             // 화페의 1/100단위를 돌려 준다
```

```
private:
        long the_credits;                    // 화페의 기본단위로
        int the_hundredths;                  // 페니로
};
```

주의: 더하기를 실현하는 메쏘드에서는 연산자로서 더하기기능을 실현하지 않고 3개의 파라메터를 가지는 돌림값을 주지 않는 (void)정적함수로서 실현한다. 실현식

```
cost = #10.50 + #5.60;
```

에 대한 코드는 다음과 같다.

```
Money const ; Money::add ( cost, Money (10,50), Money (5,60) );
```

이 실현방법을 선택한 리유에 대하여서는 15.8.3에서 보여 준다.

15.8.1 구축자앞붙이 explicit

구축자명세부에서 앞붙이 explicit는 클라스 Money의 구체례를 생성할 때 호환변환값주기가 진행되지 않도록 한다. 그러나 구축자를 호출하는데는 직접 영향을 미치지 않는다. 실례로 Money구체례에 대한 10L의 값주기는 할수 없다.

```
Money to_spend = 10L;          // 오유
                               // Money에서 long변환을 할수 없다
 Money to_spend( 1, 'a' );     // 콤파일러에서는 진행된다(그러나 애매하다)
```

주의: to_spend(1,a)에서 Money의 구축자호출을 애매하게 사용하였으므로 콤파일이 끝나지 못한다.

15.8.2 새로운 클라스 Money의 실현부

다음의 메쏘드들은 내부전개(inline)기능으로 실현한다.
따라서 그 함수들은 클라스의 명세부가 있는 파일안에 정의된다.

```
inline Money::Money( const long credits, const int pence )
{
        the_credits = credits;    the_hundredths = pence;
}
inline long Money::units( ) const
{
        return the_credits;
}
inline int Money::hundredths( ) const
{
        return the_hundredths;
}

#endif
```

분할콤파일되여야 할 Money클라스의 나머지실현부는 다음과 같다.

```
#ifndef CLASS_MONEY_IMP
#define CLASS_MONEY_IMP

void Money::add( Money& res, Money lhs, Money rhs )
{
        res.the_hundredths = lhs.the_hundredths + rhs.the_hundredths;
        res.the_credits    = res.the_hundredths / 100;
        res.the_hundredths = res.the_hundredths % 100;
        res.the_credits    = res.the_credits +
                             lhs.the_credits + rhs.the_credits;
}

ostream&  operator << (  std::ostream& s, const Money m  )
{
        s << "#" << m.units()
          << "." << (m.hundredths() < 10 ? "0" : "" )
          << m.hundredths();
        return s;
}

#endif
```

파생클라스 Pounds의 명세부는 다음과 같다.

```
#ifndef CLASS_POUNDS_SPEC
#define CLASS_POUNDS_SPEC

class Pounds : public Money {
public:
        static void prelude( const double, const double );
        explicit Pounds( const long = 0L, const int = 0 );
        operator Dollars() const;                          // 변환연산자 Dollars
        operator Francs() const;                           // 변환연산자 Francs
        friend std::ostream& operator << ( std::ostream& s, Pounds m );

private:
        static double the_dollar_rate;                     // 딸라비률
        static double the_franc_rate;                      // 프랑비률
};

#endif
```

계승에 의해 Money의 모든 메쏘드들은 새로운 클라스 Pounds에 포함된다. 보충되여야 할 메쏘드는 스위스프랑과 딸라에 대한 변환연산자, 몇개의 구축자들, 다중정의된 출력연산자 <<이다.

주의: 변환연산자명세부와 다중정의된 연산자명세부의 구조는 비슷하다.

Pounds클라스에서 구축자들을 위한 코드, prelude함수와 다중정의된 연산자 <<의 코드는 다음과 같다.

```
#ifndef CLASS_POUNDS_IMP
#define CLASS_POUNDS_IMP

#include <math>

double Pounds::the_dollar_rate = 1.0;
double Pounds::the_franc_rate = 1.0;

Pounds::Pounds( const long pounds, const int pence ) :
        Money( pounds, pence )
{
}

void Pounds::prelude( const double dollar_rate, const double franc_rate)
{
        the_dollar_rate  = dollar_rate;
        the_franc_rate   = franc_rate;
}

std::ostream& operator << ( std::ostream& s, Pounds m )
{
        s << "£" << m.units() << "."
                << (    m.hundredths() < 10 ? "0" : "" )
                << m.hundredths();
        return s;
}
```

아래의 변환연산자는 Pounds클라스에 들어 있는 값을 각각 딸라와 스위스프랑으로 변환한다.

```
Francs::operator Dollars( ) const
{
        long cents = (long) floor ( the_dollar_rate * hundredths() +
                                        the_dollar_rate * units() * 100.0 + 0.5 )
        long pounds = cents / 100;
        cents         = cents % 100;
        return Dollars( dollars, cents );
}

Francs::operator Pounds() const
{
        long pence = (long) floor ( the_pound_rate * hundredths() +
                                        the_pound_rate * units() * 100.0 + 0.5 );
```

```
        long pounds = pence / 100;
        pence          = pence % 100;
        return Pounds( pounds, pence );
}

#endif
```

이 코드는 Pounds클라스구체례가 Dollars 혹은 Swiss Francs형객체로 넘겨야 할 때마다 호출된다.

주의: 여기서는 서고함수 floor(<maths>에 있는 함수)를 사용한다. 이것은 double형 인수보다 작은 제일 큰 옹근수를 넘겨 준다. 스위스프랑과 쌍팀 등을 표현할수 있는 새로운 형을 정의하기 위해 typedef를 사용하는것이 더 좋다.

클라스 Francs와 Dollars에 대한 코드도 우와 같은 방법으로 작성한다.

15.8.3 연산자 +대면부를 메쏘드에 제공

Pounds, Dollars 혹은 Swiss Francs의 추가대면부는 그리 세련되지 못하게 실현되였다. 실례로 Pounds클라스를 쓰는 사용자는 다음과 같이 써야 한다.

```
int main()
{
        Pounds camera( 60, 0 );
        Pounds bag( 10, 0 );
        std::cout << "Cost of camera + bag is "
        Pounds res; Pounds::add( res, camera, bag );
        std::cout << res << "\n";
        return 0;
}
```

본보기함수

```
template <class Type>
inline Type operator +( Type lhs, Type rhs )
{
        Type res;
        Type::add( res, lhs, rhs );
        return res;
}
```

를 리용하면 Pounds클라스를 다음과 같이 쓸수 있다.

```
int main()
{
        Pounds camera( 60, 0 );
```

```
        Pounds bag( 10, 0 );
        std::cout << "Cost of camera + bag is " << camera + bag << "\n";
        return 0;
}
```

우의 본보기함수에 다음의 본보기함수를 추가하면 Pounds구체례에서 옹근수값을 더할수 있다.

```
template <class Type>
inline Type operator +( Type lhs, int rhs )
{
        Type res;
        Type::add( res, lhs, Type(rhs) );
        return res;
}

template <class Type>
inline Type operator +( int lhs, Type rhs )
{
        Type res;
        Type::add( res, Type(lhs), rhs );
        return res;
}
```

주의: 사용자가 다음과 같이 쓴다면 오유통보문이 나온다.

```
object = object + 1;
```

여기서 obiect는 클라스 obiect의 구체례인데 다음과 같은것을 제공하지 않았다.

- operator + 다중정의
- add(obiect, obiect, obiect)라는 함수

그리고 여기서 obiect는 obiect의 구체례에 옹근수를 돌려 주는 구축자이다.

15.8.4 종합서술

다음의 프로그람은 클라스 Dollars, Pounds, Francs에서 변환연산자들의 사용법을 보여 준다. 먼저 여러가지의 화페변환비률들을 설정한다.

```
void process()

int main()
{
        const double Dollar_Pound_rate = 0.6105;        // Dollars ->Pound
        const double Dollar_Franc_rate = 1.4613;        // Dollars -> Swiss Francs

        Dollars::prelude( Dollar_Pound_rate, Dollar_Franc_rate );

        // The money market would have precise rates which would
        // allow a small profit on conversions.
```

```
        const double Pound_Dollar_rate = 1 / Dollar_Pound_rate;
        const double Pound_Franc_rate = Pound_Dollar_rate * Dollar_Franc_rate;

        Pounds::prelude( Pound_Dollar_rate, Pound_Franc_rate );

        const double Franc_Dollar_rate = 1 / Pound_Franc_rate * Pound_Dollar_rate;
        const double Franc_Pound_rate = 1 / Pound_Franc_rate;

        Francs::prelude( Franc_Dollar_rate, Franc_Pound_rate );

        process();
        return 0;
}
```

다음의 변환들은 100.00딸라가 스위스프랑과 영국파운드로 얼마만한 가치가 있는가를 보여 준다.

```
void process()
{
        Dollars to_pay( 100.0 );
        cout << "Account to pay    = " << to_pay << "\n";
        cout << "Account to pay    = " << (Pounds) to_pay << "\n";
        cout << "Account to pay    = " << (Francs) to_pay << "\n";
}
```

클라스 Pounds, Dollars, Francs와 함께 콤파일하면 결과는 다음과 같다.

```
Account to pay   =   $100.00
Account to pay   =  ￡61.05
Account to pay   = SF146.13
```

각이한 피차(생과자일종)를 계산하는 다음의 수정프로그람에서 일반연산자 +를 다음과 같이 사용한다.

```
int main()
{
   Dollars ham_pizza( 4, 75 );
   Dollars extra_cheese = Dollars( 0, 50 );
   Dollars tuna_pizza = ham_pizza + 1;

   std::cout << "A ham pizza costs " << ham_pizza << "\n";
   std::cout << "A ham pizza wit extra cheese costs " <<
                  ( ham_pizza + extra_cheese ) << "\n";
        std::cout << "A tuna pizza costs " << tuna_pizza << "\n";
        return 0;
}
```

클라스 Dollars와 본보기함수 operator +와 함께 콤파일하면 다음과 같다.

```
A ham pizza costs $4.75
A ham pizza with extra cheese costs $5.25
A ham pizza costs $5.75
```

15.9 객체배렬의 초기화

클라스의 구축자가 파라메터(례를 들어 Dollars클라스에서처럼)들을 가질 때 이 클라스의 객체배렬을 다음과 같이 초기화할수 있다.

```
Dollars prices[4] = { 1, Dollars(2, 50), 3 };
```

주의: 초기값들을 모두 주지 않으면 기정구축자(이 경우 Dollars())가 사용된다. 구축자는 Dollars형객체에 int 1과 int 3을 넘긴다.

15.9.1 계승연산자

성원연산자들의 속성을 표로 작성하면 다음과 같다.

연산자	계승되는가	기정값에 의해 창조되는가
( ) [ ] ->	된다.	안된다.
=	안된다.	된다(성원 대 성원복사).
나머지(주의항목을 보시오)	된다.	안된다.
변환	된다.	안된다.

주의: 연산자 new와 delete에 대하여서는 17장에서 서술하며 연산자 =의 다중정의에 대하여서는 23장에서 서술한다.

15.10 자체평가

- C++에서 어떤 연산자들을 새로운 의미로 다중정의할수 있는가?
- 연산자 +는 두 연산수들이 int형일 때 새로운 의미를 가지도록 다중정의될수 있는가?
- 사용자가 새로운 연산자들을 만들어 낼수 있는가? 실례로 사용자가 어떤 옹근수에 2를 더하는 단항연산자 +++를 정의할수 있는가?
- 표준연산자를 다중정의에 많이 사용하면 프로그람작성이 왜 힘들어 지는가?

15.11 련 습

다음의 클라스들을 작성하시오.

- Imperial_Weight

  이전 영국단위인 폰드와 온스로 된 무게를 보관하는 클라스.

  이 체계에서 1폰드는 16온스이다. 따라서 코드토막

```
Imperial_Weight apples          =    Imperial_Weight( 2, 4 );
Imperial_Weight oranges    =    Imperial_Weight( 3, 14 );
 std:: out << "Combined weight is: " << apples + oranges << "\n";
```

의 결과는 다음과 같다.

```
Combined weight is: 6 Pounds 2 ounces
```

- 수자

  200자리의 정확도로 옹근수가 들어 있는 수클라스.

  이 코드를 다음과 같이 쓸수 있다.

```
Number large = Number("1_000_000_000_000_000_000_000_000_000");

large = large + 1;
```

이 클라스의 구축자는 긴옹근수 혹은 옹근수문자렬형파라메터를 가질수 있다. 만일 문자렬이 수를 초기화하는데 쓰인다면 밑선문자를 사용하여 수자들을 갈라 줄수 있다. 그렇게 하면 수를 더 쉽게 읽을수 있을것이다. 이 문제를 풀기 위한 한가지 방법은 한 배렬안에 20진형식의 수가 들어 있게 하는것이다. 다중정의연산자 +는 그때 두 연산자들의 매 자리를 개별적으로 함께 더하여 결과를 얻는다.

16 다형성

지금까지는 통보문이 객체에 보내질 때 실행되는 메쏘드가 콤파일시에 결정될수 있다고 설명하였다. 이것을 정적맺기(static binding)라고 한다. 그러나 만일 콤파일시에 객체형이 알려 지지 않는다면 메쏘드와 통보문사이의 결합은 실행시에 진행된다. 동적맺기는 다형성을 초래하는데 이것은 객체에 보내진 통보문이 객체형에 따르는 메쏘드를 실행할 때 일어 난다.

16.1 청사의 내부방들

청사에는 그림 16-1과 같은 형태의 방들이 있을수 있다.

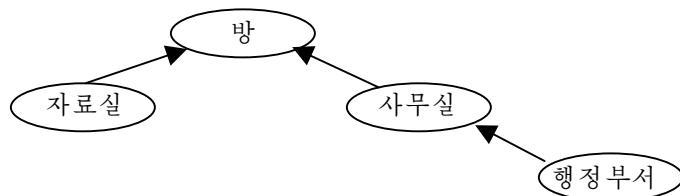



그림 16-1. 청사의 방형태

청사의 방형태는 C++계승방법을 리용하여 모형화할수 있다. 먼저 일반방을 서술하는 Room클라스를 창조한다. 그다음 이 클라스를 기초클라스로 하여 보다 세부화된 방형태를 표현하는 파생클라스계렬을 만들수 있다. 실례로 행정부서(executive room)는 벽에 풍경화도 걸려 있는 방일것이다.

Room클라스로부터 파생된 매 클라스들은 그 방에 대한 정보를 돌려 주는 describe함수를 가진다.

프로그람은 임의의 방형태구체례에 통보문 describe를 보내여 해당한 코드를 실행할수 있다. 이것은 다중정의함수를 리용하여 완성할수 있다.

그림 16-2는 Room에서 파생된 클라스구체례의 describe함수호출을 보여 준다.

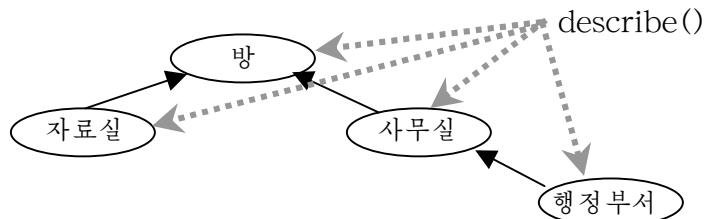



그림 16-2. Room에서 파생된 객체구체례에서의 describe함수호출

16.1.1 동적맺기

지금까지 설명한 객체에서 통보문과 메쏘드사이의 결합은 콤파일시에 진행되였다. 객체에로의 통보문보내기를 그림 16-3에서 보여 준다.

객체

```
std::string describe()
{
   return the_use;
}
-----------------
the_use
```

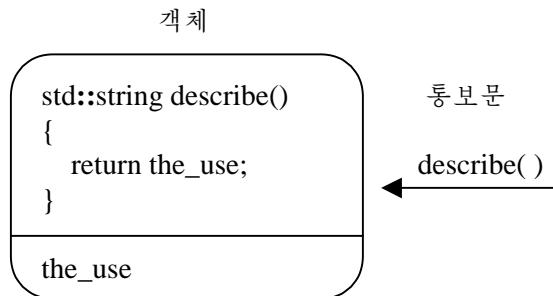
통보문

describe( ) →

그림 16-3. 객체에 통보문 describe를 보내기

통보문과 메쏘드의 결합은 객체형이 콤파일시에 알려 지지 않는다면 실행시에 제공될수 있다. 이것을 동적맺기(dynamic binding)라고 한다. 동적맺기는 어떤 객체가 각이한 형의 객체들이 모인 집합(이종집합)의 한 성원일 때 혹은 두 객체가 참조에 의한 함수호출인 경우에 발생한다. 이 두가지 경우들에 대한 정확한 설명은 후에 하기로 한다.

객체에 통보문을 보내면 객체클라스의 성원함수(메쏘드를 표현하는)를 호출한다. 메쏘드에 대한 동적맺기는 함수선언앞에 예약어 virtual을 붙여 표시한다.

16.2 Office클라스와 Room클라스

Room은 다음의 책임들을 가진다.

메쏘드	책 임
describe	방에 대한 문자렬을 넘겨 준다.
room_number	방번호를 넘겨 준다.
use	사용하는 방을 넘겨 준다.
Room	어떤 방의 상태정보를 설정한다.

이 클라스에 대한 C++명세부는 다음과 같다.

```
#ifndef CLASS_ROOM_SPEC
#define CLASS_ROOM_SPEC

#include <string>                                    // C++문자렬클라스

class Room {
public:
```

```
        Room( const int = 0, const std::string = " " );
        virtual   ~Room( );
        virtual std::string describe( ) const;              // 방에 대한 정보
        int room_number( ) const;                           // 방번호
        std::string use( ) const;                           // 사용하는 방
private:
        int     the_room_number;                            // 방번호
        std::string the_use;                                // 방의 사용
};
#endif
```

성원함수 describe와 해체자 ~Room에는 예약어 virtual을 붙인다. 예약어 virtual은 성원함수들이 호출될 때 콤파일러가 동적맺기를 리용하도록 한다. 이에 대하여서는 새로운 클라스가 Room클라스로부터 파생될 때 명백히 알려 진다.

주의: 클라스의 메쏘드가 virtual로 서술될 때 그 클라스의 해체자도 virtual로 서술되여야 한다. 그 리유에 대해서는 16.4.2에서 서술한다.

클라스 Room의 실현부는 다음과 같다.

```
#ifndef CLASS_ROOM_IMP
#define CLASS_ROOM_IMP

Room::Room( const int number,   const std::string use )
{
        the_room_number = number;
        the_use         = use;
}

Room::~Room( )
{
}

int Room::room_number( ) const
{
        return the_room_number;
}

std::string Room::use() const
{
      return the_use;
}

std::string Room::describe( ) const
{
        return the_use;
}

#endif
```

Office의 책임은 Room책임외에 다음의 내용들을 더 포함한다.

메쏘드	책 임
descibe	사무실(office)에 대한 문자렬을 돌려 준다.
Office	어떤 사무실의 상태정보를 설정한다.

Office클라스의 명세부는 Room클라스명세부의 내용과 함께 다음의 내용들을 더 포함한다.

```
#ifndef CLASS_OFFICE_SPEC
#define CLASS_OFFICE_SPEC

#include <string>                                    // C++문자렬클라스
#include <Room.h>

class Office : public Room {
public:
        Office(const int=0, const std::string="", const int=0 );
        std::string describe() const;
        int occupiers() const;
private:
        int the_occupiers;                           // 방의 인원수
};
#endif
```

Office클라스의 실현부에서 Office의 구축자는 Room의 구축자를 호출한다. 이것은 물리적인 방번호이며 그것은 Room의 구축자에 넘겨 진다.

```
#ifndef CLASS_OFFICE_IMP
#define CLASS_OFFICE_IMP

#inclued <iostream>                                  // 표준입출력
#include <strstream>

Office::Office( const int number, const std::string use,
                const int occupiers ) : Room( number, use )
{
        the_occupiers = occupiers;
}
```

describe함수는 사무실에 대한 정보를 돌려 주는 새로운 의미로 다중정의(override)된다.

```
std::string Office::describe( ) const
{
        const int MAX_BUF = 100;                    // 본문의 최대크기
        char buf[MAX_BUF];                          // 본문내용을 보관
        std::ostrstream s( buf, MAX_BUF );          // s는 문자렬흐름
        s << use();
        s << " occupied by " << the_occupiers;
        s << " people ";
        s << '\0';
        return std::string( buf );
}
```

occupiers함수는 사무실인원수를 돌려 준다.

```
int Office::occupiers( ) const
{
        return the_occupiers;
}
#endif
```

주의: 문자렬흐름클라스 ostrstream을 사용하는데 그것은 사무실정보를 서술하는데 쓰인다.

16.2.1 종합서술

우의 클라스들을 결합하여 청사의 여러 방들과 사무실에 대한 정보를 인쇄하는 프로그람을 만들수 있다. 이 프로그람은 다음과 같다.

```
void about( Room &accommodation )
{
        std::cout << "Room " << accommodation.room_number() << " : ";
        std::cout << accommodation.describe();
        std::cout << "\n";
}

int main()
{
        Room          w420( 420, "Reception" );
        Office        w414( 414, "QA", 4 );

        about( w420 );
        about( w414 );
        return 0;
}
```

about함수는 한개의 파라메터로서 Room 아니면 Office구체례를 가질수 있다. about함수의 형식파라메터는 Room&로 서술한다. Room&형의 형식파라메터는 Room의 구체례나 Room으로부터 직접 혹은 간접적으로 파생된 어떤 형의 구체례와 정합된다. C++의 엄격한 형검사에서 이러한 양보는 실행시 통보문과 메쏘드를 결합할수 있게 한다. 객체에 대한 참조는 객체의 기억주소로 된다.

about함수에서 메쏘드 describe()호출은 콤파일시에 해석될수 없다. 왜냐하면 형식파라메터 place를 넘겨 주어 표시되는 객체는 Room의 구체례일수도 있고 Office의 구체례일수도 있기때문이다. 실행시 place의 형이 실행프로그람에 의해 알려 지는 경우에는 이 호출이 해석될수 있다. 따라서 Room클라스에 있는 describe함수 아니면 Office클라스에 있는 describe함수가 호출된다.

주의: 동적맺기를 실현하기 위하여 객체에 어느 describe함수를 호출하는가에 대한 정보를 넣는다. 실행시 이 정보를 보고 해당한 describe함수를 실행시킨다. 동적맺기를 최적화하여 간접조작시간(overhead)을 2~3기계주기로 줄여야 한다.

프로그람의 실행결과는 다음과 같다.

```
Room 420 : Reception
Room 414 : QA occupied by 4 people
```

16.3 이종객체집합

다형성의 실지효과는 련관항목이 들어 있는 이종집합(heterogeneous collection)이 만들어 질 때 생긴다. 실례로 청사에서 방에 대한 정보를 가지고 있는 프로그람은 배렬을 사용하여 각이한 형태의 방들을 서술하는 객체를 넣을수 있다. 그러나 이 방법은 C++로 직접 실현할수 없다. 왜냐하면 집합의 개별적성원들의 크기가 변할수 있기때문이다. 그리하여 C++에서는 방을 표시하는 각이한 종류의 객체들에 대하여 지적자배렬을 리용한다. C++에서 지적자는 보통 참조된 객체의 물리적기억주소이다. Room과 Office형객체에 대한 지적자배렬을 그림 16-4에서 보여준다.

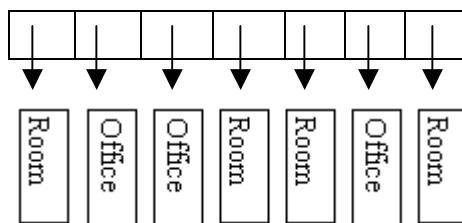



그림 16-4. Room들과 Office들의 이종집합

16.3.1 이종집합배렬

각이한 형태의 방에 대한 이종집합은 배렬로 모형화될수 있다. 배렬에는 각이한 방형태에 대해 Room의 구체례 혹은 Office의 구체례에 대한 지적자가 들어 있다.

실례로 다음의 배렬선언은 Room의 구체례 혹은 Room에서 파생된 어떤 클라스의 구체례에 대한 지적자들을 취급할수 있는 배렬 the_rooms를 정의한다.

```
Room *the_rooms[MAX];                    // 이종집합
```

주의: C++에서 Room*형객체에는 Room구체례에 대한 지적자 혹은 Room으로부터 파생된 클라스구체례인 어떤 객체에 대한 지적자를 값주기할수 있다.

이종집합은 개별적요소에 청사에 있는 방형태들을 표시하는 여러가지 객체들에 대한 배렬지적자들을 값주기하여 만든다. 실례로 420호실에 대한 정보를 이종배렬로 기입하기 위하여 다음의 코드를 사용한다.

```
the_rooms[0] = new Room( 420, "Reception" );
```

new연산자는 Room형의 클라스상수에 대하여 기억기를 할당한다. 이 기억기는 delete연산자를 사용하여 명백히 돌려 주어야 한다. 만일 그렇게 하지 않으면 프로그람이 끝날 때까지 기억기가 해제되지 않는다.

16.4 청사정보프로그람

청사내부방에 대한 정보를 보관하거나 꺼내는 용기(container)로서 사용되는 Building클라스는 다음의 책임들을 가진다.

메쏘드	책 임
add	방에 대한 정보를 추가한다.
about	지정된 방에 대한 정보를 돌린다.

Building클라스의 C++명세부는 다음과 같다.

```
#ifndef CLASS_BUILDING_SPEC
#define CLASS_BUILDING_SPEC

#include <string>

class Building {
public:
        Building();
        ~Building();
        std::string about( const int ) const;          // 방정보
        void add( Room* );                             // 방추가
private:
        static const int MAX=20;
        Room *the_rooms[MAX];                          // 이종집합
```

```
        int         the_next_free;                              // 다음번 빈방
};
#endif
```

클라스의 구축자는 간단히 배렬 the_rooms의 다음번 빈 방 the_next_free를 0으로 설정한다.

```
#ifndef CLASS_BUILDING_IMP
#define CLASS_BUILDING_IMP
#include "Building.h"

Building::Building()
{
        the_next_free = 0;
}
```

집합은 new연산자에 의해 요구된 기억기에 표시되므로 그 기억기를 delete연산자로 명백히 돌려 주어야 한다. Building클라스에 대한 해체자는 다음과 같다.

```
Building::~Building( )
{
        for ( int i=0; i < the_next_free; i++ )         // 기억기를 돌려 준다
        {
                delete the_rooms[ i ];
        }
}
```

add함수는 배렬에서 다음번에 기억할수 있는 위치에 새 자료를 추가한다.

```
void Building::add(Room* a_room )
{
        if   (the_next_free < MAX )
        {
                the_room[ the_next_free++ ] = a_room;           // 예약
                return;
        }
        throw std::range_error ( "No room" );
}
```

주의: 례외 range_error는 배렬에 빈 공간이 더는 없을 때 발생된다.

about함수는 선형탐색을 리용하여 선택된 방번호를 찾는다. 만일 방번호가 없으면 문자렬본문 “Sorry room not known”을 돌려 준다.

```
std::string Building::about( const int room ) const
{
        if    ( the_next_free > 0 )
        {
                for    ( int i=0; i<the_next_free; i++ )
                {
                        if    ( the_room[i]->room_number() == room )
                                return the_rooms[i]->describe();
                }
        }
        return "Sorry room not known";
}

#endif
```

16.4.1 종합서술

손님들이 건물에 있는 매방들에 대하여 정보를 열람할수 있도록 프로그람을 작성하는데 Room, Office, Building클라스들을 사용할수 있다. 이를 위한 간단한 시험프로그람은 다음과 같다.

```
void populate( Building& block)
{
        block.add( new Room( 420, "Reception" ) );
        block.add( new Office( 414, "QA", 4 ) );
}

int main()
{
        try
        {
                Building watts;
                int        room;
                populate( watts );                              // 일반청사
                while ( std::cout << "Room number: ",
                        std::cin >> room,
                        !std::cin.eof() )
                {
                        std::cout << "Room " << room << " : "
                                << watts.about( room ) << "\n";
                }
        }
        catch ( std::range_error& err )
        {
                std::cout << "\n" << "Fail: " << err.what() << "\n";
```

```
        }
        return 0;
}
```

프로그람을 사용하여 다음과 같이 대화할수 있다.

```
Room number:  414
Room 414 : QA occupied by 4 people
Room number: 420
Room 420 : Reception
Room number: 500
Room 500 : Sorry room not known
```

16.4.2 해체자와 다형성

클라스항목의 구체레에 대한 기억기가 해제될 때 그 클라스와 그의 모든 기초클라스들에 대한 해체자가 호출된다. 실례로 Offiice클라스의 구체레에 대한 기억기가 해제될 때 Room에 있는 해체자가 호출된다.

클라스의 구체레를 new로써 동적으로 창조할 때와 객체에 대한 지적자에 기초클라스에 대한 지적자형의 항목을 값주기할 때 기초클라스해체자를 반드시 virtual로 선언하여야 한다. 그렇지 않으면 그 클라스의 구체레를 해제할 때 기초클라스의 해체자만이 호출되게 된다. 클라스구체레창조와 관련된 지적자와 동적기억에 대하여서는 다음장에서 설명한다.

가장 간단한 방법은 가상(virtual)성원함수를 가진 기초클라스의 해체자를 virtual로 선언하는것이다.

16.5 다형성의 우결함

우점

프로그람에 대한 추가와 변경이 간단하다. 실례로 8.10에서 본 은행구좌프로그람에서 새로운 형의 구좌는 간단히 새로운 파생클라스를 창조하여 만들수 있다. 구좌에 대한 일반적업무처리를 취급하는 코드는 고정시킨다.

본질상 다형성은 프로그람에 대한 변화내용을 새로운 클라스로 실현하여 기초클라스와 함께 교갑화한다는것을 의미한다. 그러므로 프로그람을 마음대로 확장하거나 늘일수 있다는것을 알수 있다.

결함

동적맺기를 할 때마다 코드간접조작시간이 생긴다. 그것은 호출되여야 할 성원함수의 기억위치(주소)가 콤파일시가 아니라 실행시에 해석되기때문이다.

가상함수를 리용할 때는 흔히 지적자들을 리용하게 되는데 이에 대해서는 다음장에서 보다 구체적으로 설명한다.

16.6 프로그람보수와 다형성

청사의 행정부서(executive office)에 대한 정보들도 표시할수 있도록 우의 프로그람을 변경시켜 다음의 변화들을 포함시킨다.

- 새로운 파생클라스 Executive_office의 창조
- 청사에 Executive_office의 구체례를 포함시키는 추가코드작성.

프로그람의 다른 요소들은 변화시키지 않아도 된다. 프로그람을 이렇게 변경하면 다음의 우점들을 찾아 볼수 있다.

- 프로그람의 지정된 부분에서만 변화가 진행된다.
- 프로그람작성자는 프로그람보수시 프로그람의 모든 내용들을 리해하지 않아도 된다.
- 보수가 더 쉬워 진다.

다형성을 리용하여 프로그람을 잘 설계하면 프로그람을 보수, 갱신할 때 드는 비용을 상당히 줄일수 있다.

16.7 클라스의 가상항목

아래의 표는 가상화될수 있는 클라스의 요소들을 보여 준다.

클라스의 항목	가상화될수 있다	클라스의 항목	가상화될수 있다
구축자	✗	변환연산자	✓
해체자	✓	성원함수	✓
연산자(주의 볼것)	✓	동료함수	✗

주의: 가상화될수 없는 new와 delete연산자들은 17장에서 서술한다.
클라스에 있는 어떤 성원함수가 가상(virtual)함수이면 해체자도 가상함수로 만들어야 한다.

16.8 자체평가

- 정적맺기와 동적맺기의 차이점은 무엇인가?
- 객체의 이종집합이란 무엇인가? C++에서 객체의 이종집합은 어떻게 만들어지고 리용되는가?
- 다형성을 리용하여 프로그람보수를 어떻게 간단히 할수 있는가?
- 파생클라스를 기초클라스로 변환할수 있는가. 기초클라스를 파생클라스로 변환할수 있는가. 이 변환들이 안전한가. 왜 그런가?

16.9 련 습

다음의것들을 작성하시오.

- 일반사무실정보에 그 사무실에 있는 성원들에 대한 정보들을 추가한 클라스 Executive_office. 실례로 다음과 같이 작성할수 있다.
  《Ms C Lord, 프로그람관리자》
- 방, 사무실, 행정부서의 정보를 포함하는 새로운 청사정보프로그람. 가능한껏 많은 코드를 재사용해 보시오.
- 각이한 형태의 은행구좌업무들을 기록하기 위한 프로그람. 실례로 프로그람은 적어도 다음과 같은 형태의 구좌들을 처리할수 있어야 한다.
  - ✧ 리자를 지불하는 저금구좌.
  - ✧ 리자를 지불하지 못하고 사용자에게 출금할수 없는 구좌.

17 지적자와 동적기억기

이 장에서는 기억기를 직접 할당하고 조작하는 C++의 저준위특징들에 대하여 서술한다. 이러한 처리는 사실 위험하며 프로그람이 예상치 못한 오유를 범하게 할수 있다. 지적자들을 잘못 사용하여 생긴 오유들에 대한 프로그람의 수정은 힘들고 시간을 랑비할수 있다.

그러나 지적자들을 정확히 사용한 프로그람코드는 지적자를 사용하지 않고 개발한 코드보다 용량이 작고 실행이 더 빠르다.

17.1 소 개

C++는 C의 저준위특징을 대부분 다 가지고 있다. 이러한 특징의 하나가 변수의 왼쪽값(lvalue)과 오른쪽값(rvalue)을 리용하여 기억기를 직접 조작하는 능력이다.

C++의 이러한 특징들은 저준위에서 수행된다. 이 연산들은 문제해결에 사용된 클라스들을 실현하기 위해 리용될뿐이다. 지적자를 잘못 사용하면 오유가 많이 생기므로 프로그람을 그대로 쓰기가 곤난할수 있다. 기억기주소를 직접 조종하기 위해 2개의 단항연산자가 사용된다.

연산자	넘겨 주는것	용 어
&item	item 기억기안의 물리적바이트주소	왼쪽값
*item	item 안의 내용을 바이트주소로 하는 기억위치에 있는 내용	오른쪽값

주의: 실례로 int형파라메터가 참조로 전달될 때는 인수(argument)가 int&로 서술된다. 이것은 콤파일러가 item의 내용이 아니라 그의 주소를 전달할것을 요구한다. 콤파일러는 기억위치내용에 접근하기 위해 함수안에 정확한 코드를 생성한다.

<table>
<tr><th colspan="3">기억기단편구간</th><th colspan="2">&와 *를 사용</th></tr>
<tr><td>바이트주소</td><td>내용</td><td>기억위치이름</td><td>index</td><td>1004를 넘겨 준다.</td></tr>
<tr><td>1000</td><td>1028</td><td></td><td>&index</td><td>1016을 넘겨 준다.</td></tr>
<tr><td>1004</td><td>2</td><td></td><td>*index</td><td>2를 넘겨 준다.
[기억위치1004의 내용]</td></tr>
<tr><td>1008</td><td>3</td><td></td><td>**tricky</td><td>42를 넘겨 준다.
[2중간접]</td></tr>
<tr><td>1012</td><td>4</td><td>cost</td><td></td><td></td></tr>
<tr><td>1016</td><td>1004</td><td>index</td><td></td><td></td></tr>
<tr><td>1020</td><td>1000</td><td>tricky</td><td></td><td></td></tr>
<tr><td>1024</td><td>0</td><td></td><td></td><td></td></tr>
<tr><td>1028</td><td>42</td><td></td><td></td><td></td></tr>
</table>

우의 표는 바이트주소 1000부터 1028까지의 기억기단편구간을 보여 준다.

명령식 'message[20] = character;'에서 message[20]은 왼쪽값을, character는 오른쪽값을 넘겨 준다. 이것을 그림 17-1에서 설명한다.

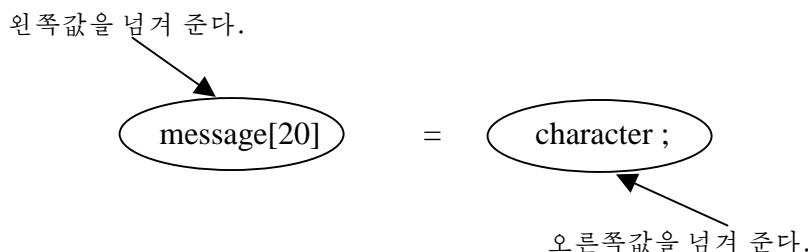



그림 17-1. C++의 오른쪽값과 왼쪽값기능

C++에서 첨수연산자는 주소계산을 리용하여 실현될수 있다.

다음의 코드에서는 벡토르를 선언하고 지적자변수에 첫번째 요소의 주소를 값주기한다.

```
const  int SIZE = 10;
char  ch_vec[SIZE];     char *p_ch = ch_vec;
int     int_vec[SIZE];     int  *p_int = int_vec;
```

주의: 벡토르의 이름은 첫번째 요소의 주소를 넘겨 준다. ch_vec와 &ch_vec[0]는 둘 다 첫번째 요소 ch_vec의 주소를 넘겨 준다.

문자벡토르와 옹근수벡토르의 4 번째 요소는 다음의 두가지중 어느 하나를 리용하여 출력된다.

배렬접근을 리용하여	주소계산을 리용하여
cout << ch_vec[4]; cout << int_vec[4];	cout << *(p_ch+4); cout << *(p_int+4);

주의: C++에서는 옹근수값이 주소에 부가되면 그 주소가 붙은 항목의 바이트크기로 맞추어 진다. int형이 4개 바이트로 표현된다면 p_int의 바이트주소에는 16이 더해 진다.

17.2 C++에서 지적자의 리용

문자 'B', 'r', 'i' 등으로 초기화된 100 개 문자의 배렬을 선언해 보자.

```
char name[100] = "Brighton East Sussex";
```

문자변수의 주소를 가지는 변수 p_ch 를 다음과 같이 선언한다.

```
char *p_ch;
```

다음의 코드는 p_ch 에 배렬 name 의 4 번째 요소주소를 기억한다.

```
char  name[100] = “Brighton East Sussex”;
char  *p_ch;

p_ch = &name[3];
```

이것을 그림 17-2 에서 보여 준다.

p_ch

name

B	r	i	g	h	t	o	n		E	a	s	t		S	u	s	s	e	x	\0

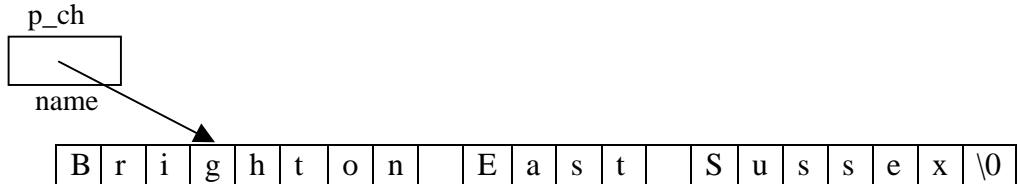



그림 17-2. name[3]의 지적자

17.3 배렬로부터 지적자에로

C++에서 C문자렬은 기억기에 기억될 때 문자들이 련속적으로 기억되고 마지막에 끝을 표시하는 특수문자 ‘\0’이 붙는다. 특수문자 ‘\0’은 수값 0을 가지고 있는데 이 0은 론리거짓값을 나타내기도 한다. C++는 이러한 기능을 제공하는 서고함수들을 가지고 있지만 C문자렬들을 조종하는 고유한 언어기능은 가지고 있지 않다.

문자렬조작의 보다 발전된 형태를 실현하는 클라스 string은 C++언어부분이 아니라 표준클라스서고이다. 부록 3은 C++문자렬조작서고함수들의 일부를, 부록 4는 클라스 string성원함수들의 일부를 서술한다.

배렬을 사용하여 어떤 기억령역으로부터 다른 기억령역에로 문자렬을 복사하는 함수를 실현해 보자.

```
void strcpy_V1( char to[ ], const char from[ ] )
{
   int   i = 0;                          // 배렬의 시작
   while ( from[i] != ‘ \0 ’ )           // EOS 가 아닌 동안
   {
      to[i] = from[i];                   // 원천지에서 목적지에로 문자를 복사
      i ++;                              // 다음문자
   }
   to[i] = ‘ \0 ’ ;                      // 목적지에 EOS 를 추가
}
```

주의: 문자렬은 문자‘ \0 ’으로 끝나는데 이 문자는 문자들의 복사를 완료한다. EOS는 문자렬끝기호‘ \0’을 의미한다.

배렬대신에 지적자를 사용한 실현부는 다음과 같다.

```
void strcpy_V2( char* to, const char* from )
{
  while ( *from != ‘ \0’ )          //  EOS 가 아닌 동안
  {
    *to = *from;                    // 원천지로부터 목적지에로 문자를 복사
     to ++;  from ++;               // 지적자들을 증가
  }
  *to = ‘ \0’;                      // 목적지에 EOS 를 추가
}
```

주의: 문자지적자를 포함하는 파라메터는 char*로 서술된다. from++는 다음항목을 지적하는 from에로 증가시키는데 이 경우 한 바이트씩 처리된다.

우의 원천코드는 from으로 지적된 문자를 넘겨 주는 표현형식 *from++를 사용하여 크기를 감소시키며 이때 다음기억위치를 지적하는 from안의 주소가 증가된다.

```
void strcpy_V3( char* to, const char* from )
{
    while ( *from != ‘ \0’ )        //  EOS 가 아닌 동안
    {
       *to ++ = *from ++;           // 지적자를 복사하고 증가
     }
    *to = ‘ \0’;                    // 목적지에 EOS 를 추가
}
```

크기의 감소는 조건 *from != ‘\0’이 *from과 같은가를 관찰하는것에 의하여 실현된다. C++에서 거짓은 0, 참은 0이 아닌 다른 값이라는것을 기억하시오.

```
void  strcpy_V4( char* to, const char* from )
{
    while ( *from )                 // EOS 가 아닌 동안
    {
       *to ++ = *from ++;           // 지적자를 복사하고 증가
     }
     *to = ‘ \0’;                   // 목적지에 EOS 를 추가
}
```

while 순환문의 본체를 조건으로 대신하는 코드는 다음과 같다.

```
void  strcpy( char* to, const char* from )
{
        while ( *to++ = *from++ );          // EOS 가 복사될 때 복사중지
}
```

주의: 문자렬끝기호 '\0' 은 끝검사가 진행되기전에 복사된다.

만일 파라메터가 등록기안에 배치된다면 최량코드는 보통 등록기파일방식의 기계에서 만들어 질것이다.

```
void  strcpy( register char* to, const register char* from )
{
        while ( *to++ = *from++ )                    // EOS 가 복사될 때 복사중지
}
```

주의: 머리부파일 <string.h>안의 C 표준서고함수 strcpy 는 목적지에 복사되는 EOS 문자의 주소를 함수결과로서 돌려 준다.

얼핏 보면 이것은 그리 명백치 않지만 C++표현형식의 한가지이다. 사용자는 문자렬조작에 대체로 문자렬클라스를 사용하므로 이에 대해서는 잘 알수 없다. 그러나 프로그람작성자들은 우의 기능을 리용한 문자렬클라스의 코드를 많이 사용한다.

이것은 필요하다면 프로그람작성자들이 저준위기계구조들에 접근할수 있도록 하는 C++의 원리를 잘 보여 주는 실례이다. 그러나 이러한 특징들은 반드시 요구될 때만 사용되며 대체로 클라스기구(class mechanism)들에 의해 숨겨 진다.

주의: 기계단어에 포함될수 있는 어떤 국부변수나 파라메터는 프로그람작성자에 의해 CPU등록기안에 포함된것으로 지적될수 있다. 이것은 선언앞에 register를 표시하여 나타낸다. 물론 콤파일러는 이러한 암시를 무시할수 있다.
서고함수 strcpy도 문자렬이 복사된 기억령역의 주소를 넘겨 준다.

17.4 지적자와 배렬

아래의 프로그람들은 문자렬안의 문자수를 계수하는 함수를 실현한다. 문자렬은 첫번째 프로그람에서는 문자배렬로, 두번째 프로그람에서는 문자지적자로 서술된다.

우선 배렬을 사용하는 프로그람은 다음과 같다.

```
#include <iostream>

namespace strlen_array
{
    int strlen( const char str[ ] )
    {
        int  position = 0;
        while( str[ position ] != '\0' )
        {
            position ++;
        }
        return( position );
    }
}
```

```
int main( )
{
    static char str[ ] = " University of Brighton" ;
    std::cout << " The length of " << str << " is " << strlen_array::strlen(str)
                                            << " characters " << " \n " ;
    return 0;
}
```

주의: strlen의 정의가 들어 있는 이름공간 strlen_array를 사용하는것은 체계이름 strlen과의 충돌을 피하기 위해서이다.

실행결과는 다음과 같다.

```
The length of University of Brighton is 22 characters
```

다음으로 문자렬안의 문자수를 계수하기 위해 지적자를 사용한 프로그람은 아래와 같다.

```
#include <iostream>
namespace strlen_ptr
{
    int strlen( const register char* str )
    {
        int count = 0;
        while ( *str++ )   count ++;
        return( count );
    }
}
```

```
int main( )
{
    static char str[ ] = " University of Brighton" ;
    std::cout << " The length of " << str << " is " << strlen_array::strlen(str)
                                     << " characters " << " \n " ;
    return 0;
}
```

실행결과는 다음과 같다.

```
The length of University of Brighton is 22 characters
```

우의 프로그람은 문자계수를 실현하는 아주 효과적인 코드로 이루어 져 있다. 대체로 지적자를 리용하겠는가 배렬을 리용하겠는가 하는것은 그 실현자에게 달려 있지만 지적자를 리용해야만 문제를 풀수 있는 경우가 있게 된다.

주의: C 뿐만아니라 C++의 설계목적의 하나는 등록기파일구조의 효률적인 기계코드를 만드는것이다. 배렬이 함수에 전달될 때 형식파라메터는 다음의 선언들중 어느 하나로 서술된다.

```
Type  *formal_parameter
Type   formal_parameter[]
```

17.4.1 지적자와 배렬을 위한 파라메터접근권

사용자는 배렬이나 지적자를 함수에 전달하여 형식파라메터로 서술된 항목들에 대한 쓰기접근을 조종할수 있다. 그러나 읽기접근은 조종할수 없으며 항상 허용된다.

이것은 다음의 6개 함수구조에 의하여 개괄적으로 볼수 있다.

```
void process( char * vec )
{
   // vec 는 char 형지적자이다
   vec ++;                     // 정확함
   vec[1] = ‘z’;               // 정확함
   *(vec + 1) = ‘z’;           // 정확함
}
```

```
void process( const char * vec )
{
   // vec 는 const char 형지적자이다
   vec ++;                     // 정확함
   vec[1] = ‘z’;               // 콤파일실패
   *(vec + 1) = ‘z’;           // 콤파일실패
}
```

```
void process( const char * const vec )
{
// vec 는 const char 형상수지적자이다
   vec ++;                     // 콤파일실패
   vec[1] = ‘z’;               // 콤파일실패
   *(vec + 1) = ‘z’;           // 콤파일실패
```

```
void process( char * const vec )
{
   // vec 는 char 형상수지적자이다
   vec ++;                     // 콤파일실패
   vec[1] = ‘z’;               // 정확함
   *(vec + 1) = ‘z’;           // 정확함
}
```

```
void print_vec4( const char vec[ ] )
{
   // vec 는 상수배렬이다
   vec ++;                     // 콤파일실패
   vec[1] = ‘z’;               // 콤파일실패
   *(vec + 1) = ‘z’;           // 콤파일실패
}
```

```
void process( char vec[ ] )
{
   // vec 는 배렬이다
   vec ++;                     // 정확함
   vec[1] = ‘z’;               // 정확함
   *(vec + 1) = ‘z’;           // 정확함
}
```

주의: 함수파라메터가 배렬이면 const char* vec로 서술할수 없다. 값주기명령문 vec[1]= ‘z’와 (vec+1)= ‘z’는 결과적으로 같다. 파라메터 vec에로의 읽기접근은 일반적으로 첨수 ‘vec[i]’나 간접연산 ‘*(vec+i)’을 사용하여 진행한다.

17.5 동적기억기할당

C++에서 기억기(storage)는 체계로부터 동적으로 요구될수 있다. 다음의 실례는 10개의 문자를 기억하는데 필요한 기억기의 지적자를 p_ch로 값주기한다.

```
char  *p_ch;
p_ch = new char[10];                    // 기억기할당
```

다음의 조작에 의하여 이 기억기는 체계에 돌려 진다.

```
delete  [ ] p_ch;                       // 기억기해방
```

주의: 괄호 []는 항목들의 배렬이 처리되는 과정을 표시하는데 사용된다. 괄호 []가 필요 없는 경우도 있지만 배렬이 클라스의 구체례들을 포함할 때 괄호 []를 생략하면 매개 요소의 해체자를 호출할수 없다.

17.5.1 다른 방법

항목이 한개라면 괄호 []는 생략될수 있다. 실례로 다음의 코드결과는 같다.

[]의 사용	[]의 비사용

`char  *p_ch;`
`p_ch = new char[1];`
`delete [] p_ch;`
	`char  *p_ch;`
	`p_ch = new char;`
	`delete  p_ch;`

주의: 만일 기억기를 할당할수 없다면 례외 bad_alloc가 발생된다. 그러나 유산콤파일러(legacy compiler)들은 NULL지적자를 돌려 준다.

17.5.2 동적기억기에로의 접근

동적기억기는 보통 함수가 없는 클라스의 기억기할당에 리용된다. 이 구조를 C언어때부터 사용한 구조체로 서술할수도 있다.

클라스정의문법을 리용	구조체정의문법을 리용

`const  int  MAX_NAME = 40;`
`class  Person {`
`public:`
`    int     birthdate;`
`    char  sex;`
`    char  name[MAX_NAME];`
`};`
	`const  int  MAX_NAME = 40;`
	`struct  Person {`
	`    int     birthdate;`
	`    char  sex;`
	`    char  name[MAX_NAME];`
	`};`

주의: 사람이름표시에는 문자렬클라스의 구체레보다 문자배렬을 사용한다. 문자배렬을 리용하여 문자렬을 표현할 때 표준서고함수들이 문자를 배렬에 복사하고 배렬에서 꺼내는데 리용된다. 17.4 에서는 문자배렬을 사용하는 방법과 문자렬들을 가지는 클라스 string 의 구체레를 사용하는 방법을 비교해 본다.

우에서 정의한 기억기는 사람에 대한것인데 다음과 같이 사용될수 있다.

```
Person father;

father.birthdate = 1918;
father.sex = ‘M’;
srecpy( father.name, “ James Smith” );
```

주의: 함수 strcpy 를 사용하여 사람이름을 이름마당안에 복사한다.

기억기 mother 의 할당을 직접적으로가 아니라 구조체지적자 p_mother 를 사용하여 간접적으로 할수 있다.

```
Person  *p_mother;
```

기억기 p_mother 는 기계에서 다음과 같다.

p_mother

주의: p_mother 는 어떤 Person 형기억기의 지적자를 가질수 있는 변수이다. p_mother 의 내용은 현재 정의되여 있지 않다.

Person 형구체레의 기억기할당은 다음과 같다.

```
p_mother = new Person[1];
```

이 자료구조에 대한 기억기의 설명을 그림 17-3 에 준다.

p_mother

birthdate sex name

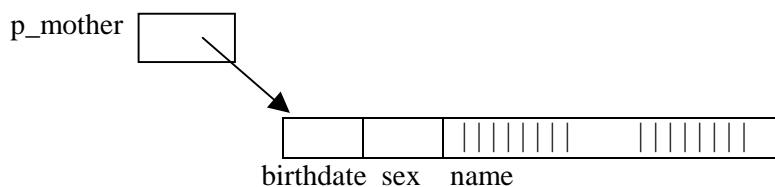



그림 17-3. p_mother = new person[1]; 의 기억기배치

주의: Person 형기억기의 1 번 요소에 할당된다.

기억기 mother 의 개별적요소들에 대한 접근은 다음과 같다.

```
p_mother ->birthdate = 1918;
p_mother ->sex = 'F';
strcpy( p_mother ->name, "Margaret Smith" );
```

주의: 클라스의 성원들에 접근하기 위해 선택연산자 ->가 사용된다. 선택연산자 ->는 클라스의 구체례가 객체지적자로 표시될 때 .대신에 사용된다.
p_mother->birthdate = 1918;를 (*p_mother).birthdate = 1918;로 표현할수 있다.

17.6 동적기억기의 사용

동적기억기할당을 리용하여 8.9에 서술한 클라스 Stack를 다시 작성해 보자. 이때 클라스 Stack의 사용자대면부는 변화되지 않는다. 따라서 이 클라스의 사용자는 자기의 원래프로그람을 수정하지 않아도 된다.

그러나 클라스의 구체례가 복사되는것이라면 클라스 Stack는 사용될수 없다. 23장에서는 기대했던 결과가 나오지 않은 리유와 복사될수 있는 동적기억기를 사용하는 클라스의 실현방법에 대하여 서술한다. 사실 클라스 Stack의 구체례를 복사하는것은 두 객체가 같은 자료를 지적하기때문이다. 이 두 객체의 매 해체자는 공유된 기억기를 해제하려고 시도할것이다.

17.6.1 클라스 Stack의 명세부

```
#ifndef  CLASS_STACK_SPEC
#define  CLASS_STACK_SPEC

#include  <ctype.h>

template <class type>
class Stack {
public:
    Stack( );
    ~ Stack( );
    bool  empty( ) const;                 // 빈 탄창
    size_t  size( ) const;                // 집합의 크기
    const Type&  top( ) const;            // 탄창의 꼭대기항목을 돌려 준다
    Type&  top( );                        // 탄창의 꼭대기항목을 돌려 준다
    void push( const Type);               // 탄창에 항목을 넣는다
    void pop( );                          // 탄창에서 꼭대기항목을 뽑는다
private:
    class Element;                        // 림시선언
    typedef  Element  *p_ Element;
    class Element  {                      // 완전한 선언
    public:
        Type          the_value;          // 기억된 항목
```

```
        p_ Element   the_next;                    // 다음요소의 지적자
    };
    p_ Element   the_p_tos;                       // 첫번째 항목의 지적자
    int   the_no_elements;                        // 탄창의 요소수
};

#endif
```

주의: P_Element 는 림시선언이다.

17.6.2 클라스 Stack 의 실현부

빈 목록을 나타내는 변수 the_p_tos 는 빈 지적자 NULL 을 가진다. 그림 17-4 에서 보는바와 같이 빈 지적자(nilpointer)는 변수 the_p_tos 가 현재 그 어떤 기억기를 지적하지 않는다는것을 나타내는데 사용된다.

the_p_tos [•]

그림 17-4. 빈 지적자(null pointer)에 대한 설명

클라스 Stack 의 구축자는 빈 지적자를 초기값으로 설정하여 탄창안의 요소수를 0 으로 설정한다.

```
#ifndef  CLASS_STACK_IMP
#define  CLASS_STACK_IMP
#include  "stack.h"

template <class Type>
Stack<Type>::Stack( )
{
   the_p_tos = NULL;                              // 빈 탄창
   the_no_elements = 0;                           // 요소가 없다
}
```

해체자는 현재 쓰이고 있는 기억기를 해방하여 체계에 돌려 준다.

```
template <class Type>
Stack<Type>::~Stack( )
{
    while ( !empty( ) ) pop( );                   // 기억기해방
}
```

주의: 성원함수 pop 는 항목이 기억된 기억기를 해방한다.

성원함수 size 는 탄창안의 요소수를 돌려 준다.

```
template  <class Type>
bool Stack <Type>::empty( ) const
{
   return  the_p_tos == NULL;                              // 비였다
}
```

성원함수 empty 는 탄창이 비였을 때 참을 돌려 준다.

```
template <class Type>
size_t  Stack<Type>::size( ) const
{
 return  the_no_elements;                                  // 탄창안의 요소수
}
```

그림 17-5 에서 보여 준 <int> stack 의 구체례안에 단정확도형옹근수값 3 이 기억될 때의 기억기할당을 그림 17-5 에 보여 준다.

the_tos

3 | •

the_value   the_p_next

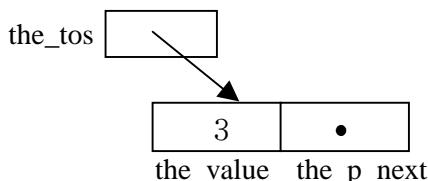



그림 17-5. 한개의 요소를 가진 탄창

the_tos 는 자료구조 Element 구체례의 지적자를 가진다. 이때 자료구조 Element 의 성원 the_value 를 호출하기 위해 다음의 코드가 사용된다.

```
the_tos -> the_value = 3;
```

함수 push 는 새로운 요소를 만들고 그것을 탄창우에 넣어 진 항목들을 가지는 요소들의 련결목록안에 련결한다.

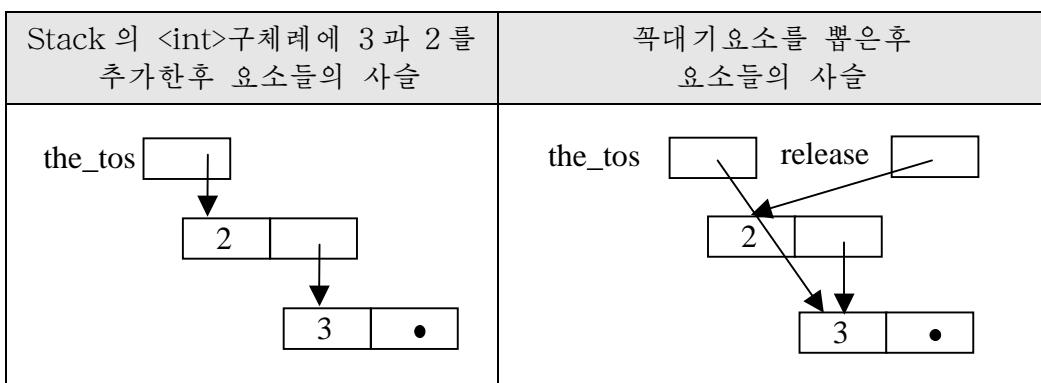
클라스 Stack의 <int>구체례에 3과 2를 추가한후의 요소들의 사슬	클라스 Stack의 <int>구체례에 1을 넣은후의 요소들의 사슬
the_tos → [2 |] → [3 | •]	the_tos → [1 |] → [2 |] → [3 | •]

함수 push 의 실현부안에 사용가능한 기억기가 없다면 레외가 발생되여 포착되고 새로운 요소의 추가실패를 알리는 레외 range_error 가 또 발생된다.

```
template <class Type>
void Stack<Type>::push(const Type item)
{
      P_Element  p_new_item = NULL;
      try
      {
           p_new_item = new Element [1];                          // 할당
       }
      catch (bad_alloc& exp)
      {
           throw std::range_error( "Stack: out of mem" );       // 실패됨
       }
      p_new_item -> the_value = item;                             // 자료기억
      p_new_item -> the_p_next = the_p_tos;                       // 사슬고리안에 련결
      the_p_tos = p_new_item;
      the_no_elements ++;                                         // 하나 증가
}
```

주의: 기억기가 사용불가능일 때 취해 진 이전의 동작에 대해서는 26.5 의 유산콤파일러 부분에서 서술한다. 레외 bad_alloc 를 포착할 때 동적기억기가 불충분하여 새로운 레외 range_error 가 발생될수 있다.

성원함수 pop 는 체계로 돌려 주는 탄창꼭대기항목의 기억기를 해방한다.

Stack 의 <int>구체레에 3 과 2 를 추가한후 요소들의 사슬	꼭대기요소를 뽑은후 요소들의 사슬
the_tos → [2 |] → [3 | •]	the_tos, release → [2 |] → [3 | •]

```
template  <class Type>
void Stack <Type>::pop( )
{
   if (the_p_tos == NULL) {                                       // 빈 탄창
       throw std::range_error( "Stack: underflow" );
   }
```

```
    P_Element   release = the_p_tos;                          // 꼭대기항목을 삭제
    the_p_tos = the_p_tos->the_p_next;
    delete [ ] release;
    the_no_elements --;                                       // 하나 감소
}
```

성원함수 top 는 탄창꼭대기항목의 내용들을 돌려 준다. 이 성원함수에는 객체의 상수, 비상수집합체를 제공하는 2 개의 부류가 있다.

```
template  <class Type>
Type& Stack <Type>::top( )
{
     if (the_p_tos == NULL)
         throw std::range_error( "Stack: underflow" );
     return the_p_tos -> the_value;                           // 꼭대기요소
}

template  <class Type>
const  Type& Stack<Type>::top( )  const
{
     if (the_p_tos == NULL)
         throw std::range_error( "Stack: underflow" );
     return the_p_tos -> the_value;                           // 꼭대기요소
}

#endif
```

17.6.3 종합서술

```
int main ( )
{
  Stack <int> numbers;
  char ch;
  try
  {
       while ( std:;cin >> ch, !std::cin.eof( ) )
       {
           swith (ch)
           {
               case  '+' :                                    // 탄창우에 항목을 넣기
               {
                     int num; std::cin>>num;
                     numbers.push(num);
                    break;
               }
```

```
        case `-`:                                   // 탄창에서 항목을 뽑기
        {
            int num = numbers.top ();
            std::cout << "Num =" << num<< "\n";
            numbers.pop();
            break;
        }
      }
    }
  }
  catch (std::range_error& err )
  {
   std::cerr << "\n"<< "Fail: " <<err. what() << " \n";
  }
  return 0;
}
```

자료

```
+1 +2 +3 +4 - - - - -
```

를 가지고 실행된 결과는 다음과 같다.

```
Num = 4
Num = 3
Num = 2
Num = 1
Fail: Stack: underflow
```

이것은 앞에서 탄창으로 사용된것과 같은 대면부이다. 그러나 여기서 탄창의 실현부코드는 동적으로 할당된 기억기를 사용한다.

17.7 구조체와 클라스

구조체(struct)는 C 언어에서부터 쓰이였으며 거기서는 자료성원들만을 가지고 있었다. 그러나 C++에서 구조체는 그의 성원들이 공개형(public)을 기정으로 하고 있다는것을 제외하고 클라스와 같은 동작을 한다. 따라서 C++에서 struct 는 성원함수들을 가질수 있다.

실례로 클라스 Element

```
class Element;
typedef Element *p_Element;
class Element {
```

```
public:
  Type        the_value;         // 보관된 항목
  P_Element   the_p_next;        // 다음요소의 지적자
};
```

는 다음의 구조체 Element 로 선언될수 있다.

```
struct  Element;
typedef  Element *p_Element;
struct   Element {
  Type        the_value;          // 보관된 항목
  P_Element   the_p_next;         // 다음요소의 지적자
};
```

구조체는 그의 성원들이 모두 공개형을 기정으로 하고 있는 클라스이다.

17.8 동적기억기할당과 정적기억기할당

다음의 표에는 동적기억기할당과 정적기억기할당을 리용하는것이 가지는 우결함을 개괄한다.

기 준	동적(련결)기억기할당	고정(순차)기억기할당
사용되는 공간(정확한 용량은 알수 없다.)	적당하다.	사용되지 않는 요소들에도 공간이 랑비된다.
사용되는 공간(정확한 용량은 알수 없다.)	련결을 위해 요구되는 추가공간	적당하다.
코드의 복잡성	기억기를 조작하는 코드가 복잡해 질수 있다.	간단하다.
구조체의 자료항목에 대한 임의접근	매우 느리게 될수 있다.	빠르다.
구조체의 자료항목에 대한 순차적접근	빠르다.	빠르다.

주의: 할당되여야 할 기억령역의 크기가 콤파일시에 고정되는 경우에는 고정기억기할당을 쓴다. 콤파일시에 그 크기가 고정되는 구조체의 실례가 바로 C++의 배렬이다.

17.9 new 와 delete 연산자의 다중정의

연산자 new 와 delete 는 클라스안에서 다중정의될수 있다. 이것은 클라스의 실현자(implementor)가 자기의 기억기처리루틴들을 제공할수 있게 한다.

다음의 클라스 Float 는 류점수를 기억하며 클라스의 구체레가 동적으로 창조될 때 특징정보들을 제공하는 new 와 delete 를 다중정의한다.

```
#ifndef   CLASS_FLOAT_SPEC
#define   CLASS_FLOAT_SPEC

class Float
{
public:
    Float (float = 0.0 );                         // 구축자
    void* operator new (size_t);                  // 단일객체
    void* operator new(size_t);                   // 객체배렬
    void  operator delete ( void*, size_t);       // 단일객체
    void  operator delete[ ] (void*, size_t);     // 객체배렬
    operator float ( );                           // 변환연산자
private:
    float the_storage;                            // 기억기
};

#endif
```

주의: 다중정의연산자 new 는 형이 정의되지 않은 기억령역(void*)에 돌려 준다.

클라스 Float 의 실현부는 다음과 같다.

```
#ifndef   CLASS_FLOAT_IMP
#define   CLASS_FLOAT_IMP

Float::Float( float  f )
{
  the_storage = f;
}
```

연산자 new 의 다중정의에는 2 가지가 있다. 하나는 단일객체를 위한 다중정의이고 다른 하나는 클라스객체배렬이 할당될 때의 다중정의이다.

```
void* Float::operator new( size_t  s )
{
   std::cout << “ [new   Bytes =”  <<s << “]” << “\n”;
   return (void*) :: new char[s];                              // 대역 new
}
void* Float::operator new [ ] ( size_t  s )
{
   std::cout<< “[new[ ]  Bytes =” << s << “ ]”<< “ \n”;
   return (void*) :: new char[s];                              // 대역 new
}
```

연산자 delete 의 다중정의에도 2 가지가 있다. 하나는 단일객체를 위한 다중정의이고 다른 하나는 클라스객체배렬이 할당되지 않을 때의 다중정의이다.

```
void Float::operator delete (void *p, size_t s)
{
   std::cout << “ [delete Bytes =”<<s<< “]” << “ \n”;
   ::delete p;                                          // 대역 delete
}
void Float::operator delete[ ] (void *p, size_t s)
{
  std::cout<< “ [delete[] Bytes =” <<s<< “]”<< “ \n”;
  ::delete [ ] p;                                       // 대역 delete
}
#endif
```

주의: 다중정의연산자 new와 delete는 암시적으로 선언되는 정적연산자이다. 이것은 이 연산자들이 클라스의 자료성원들에 접근하지 않는다는것을 의미한다. 다중정의연산자 delete의 파라메터는 삭제되여야 할 개별적인 기억기의 단위의 크기를 바이트단위로 제공한다. size_t는 머리부파일 <stddef.h>안에 옹근수값으로 정의된다. void*은 어떤 형의 기억기에 대한 지적자를 가리킨다.

클라스 Float는 류점수사용에 대한 특징정보를 주는데 리용된다.

```
int main ()
{
   std::cout << “ Float*s1 = new Float (3.14);” << “ \n”;
   Float *s1 = new Float(3.14);                    // 클라스 Float 의 기억기할당
   std::cout<< “ Foat *s2 = new Float [10];” << “ \n”;
   Float *s2 = new Float [10];                     // 클라스 Float 의 기억기할당

   std::cout<< “ delete s1;”<< “ \n”;

   std::cout<< “ delete s1;” << “ \n”;
   delete [ ] s1;                                  // 클라스 Float 의 기억기해제
   std::cout << “ delete [ ] s2;” << “ \n”;
   delete [ ] s2;                                  // 클라스 Float 의 기억기해제
   return 0;
}
```

실행결과는 다음과 같다.

```
Float *s1 = new Float (3.14);
 [new    Bytes   = 4]
```

```
Float *s2 = new Float [10];
  [new[] Bytes  = 40 ]
Contents of s1 = 3.14
delete s1;
  [delete Bytes = 4 ]
delete [] s2;
 [delete [] Bytes = 4 ]
```

17.9.1 대역연산자 new와 delete의 다중정의

대역연산자 new와 delete도 다중정의된다. 그러나 많은 서고함수들이 new와 delete를 사용하므로 심중히 써야 한다. 특수하게 C++입출력체계에서는 동적으로 할당된 기억기를 사용하여 완충기를 할당한다.

17.9.2 연산자 new의 추가파라메터

연산자 new에는 추가파라메터들이 넘겨 질수 있다. 이런 형식으로 사용될 때 new는 다음과 같이 서술된다.

new의 서술	**클라스 Type의 호출실례**
void* operator new (size_t, parameters)	new (parameters) Type;
void* operator new[ ] (size_t, parameters)	new (parameters) Type[5];

new연산자의 이러한 형식은 현재의 기억기우에 객체를 만드는데 쓰인다.

실례로 아래에서 연산자 new의 다중정의는 그 객체를 위하여 할당된 기억기로서 두번째 실제파라메터를 돌려 준다.

```
void* operator new (size_t, void* storage)
{
   return  storage;
}
void* operator new[] ( size_t, void* storage )
{
   return  storage;
}
```

다중정의연산자 new가 실행된후에 객체의 구축자는 암시적으로 호출된다. 다음의 코드는 우의 연산자 new의 다중정의를 사용하여 현재의 기억기우에 클라스 Office(16.2에서 서술된다.)의 구체례를 할당한다.

```
int main ()
{
   char *p_object  = new char [ sizeof(office) ];          // 초기의 기억기
```

```
    Office *p_office) = reinterpret_cast<Office>(p_object);
    new( p_office) Office(420, "QA", 3);              // 구축자호출
    std::cout << p_office -> describe( ) << "\n";
    p_office -> ~office( );                            // 해체자호출
    delete [ ] p_object;                               // 기억기를 돌려 준다
    return 0;
}
```

주의: 우와 같은 연산자 new를 사용하여 할당된 클라스 Office의 구체례에 대하여서는 해체자가 명시적으로 호출된다. 그 리유는 콤파일러가 기억기에 문자들이 기억되여 있다고 생각하고 Room의 구체례에 대한 해체자를 자동적으로 호출하지 못하기때문이다. reinterpret_cast는 지적자를 char *로부터 office *로 강제형변환하는데 리용된다.

우의 코드는 초기의 기억기를 클라스객체를 위한 기억기로 전환시켜야 할 때 리용된다. 20.2.6에서는 이에 대한 사용실례를 설명한다.

17.10 표준연산자 new와 delete

아래의 표는 머리부파일 <new>에서 정의된 new와 delete의 표준선언을 개괄한다. new나 delete는 클라스성원함수로 정의되면 정적성원으로 볼수 있는데 그것들은 계승되기는 하지만 가상화될수는 없다.

연산자	선 언	주의

new	void* operater new (size_t size) throw(bad_alloc);
	void* operater new[ ] (size_t size) throw(bad_alloc);
	void* operater new (size_t size, const nothrow&) throw( );
	void* operater new[ ] (size_t size, const nothrow&) throw( );
	void* operater new (size_t size, void* p) throw( );
	void* operater new[ ] (size_t size, void* p) throw( );
		[1]
		[1]
		[2]
		[2]
		[2]
		[2]

delete	void* operater delete (void* p, void*) throw( );
	void* operater delete (void* p) throw( );
	void* operater delete[ ] (void* p) throw( );
	void* operater delete[ ] (void* p, void*) throw( );
		[3]

주의: [1] 기억기를 할당할수 없는 경우 례외 bad_alloc를 내보낸다.
[2] 기억기를 할당할수 없는 경우 빈 지적자를 돌려 준다.
[3] delete는 둘이상의 정의를 가지도록 다중정의할수 없으며 size_t는 머리부파일 ctype.h에서 정의된다.

17.10.1 기억기할당실패

기억기가 고갈되면 체계는 이 우발적인 사건에 대한 조종자를 호출한다. 이 조종자함수(handler function)는 보통 new가 현재요구를 만족시키도록 하기 위하여 더미령역(heap)에로 기억기를 돌려 주게 된다. 조종자는 원형이 다음과 같이 머리부파일 <new>에서 정의된 함수 set_new_handler를 호출하는것에 의하여 체계에 넘겨 진다.

```
typedef void (*new_handler( );
new_handler set_new_handler(new_handler fun);
```

조종자에 대한 기정값은 NULL 인데 이 값은 그 어떤 함수도 호출될수 없다는것을 가리킨다. new 연산자가 호출될 때 모조코드에서는 다음의 처리가 수행된다.

```
while( true )
{
      allocate_memory( );
      if ( memory_is_allocated )  return pointer_to_memory;
      if ( new_handler == Null )
      {
        if ( no_throw ) return NULL;  else throw bad_alloc ( “ ” );
      }
      new_handler( );
}
```

주의: 연산자 new 가 례외를 내보내는가 빈 지적자를 돌려 주는가에 따라 서로 다른 동작이 수행된다.

17.11 연산자 .*과 ->*

때때로 주소를 통하여 클라스성원을 참고하는것이 유익한 경우도 있다. 그것을 정확히 수행하자면 특별한 처리가 리용되여야 한다.

이 처리는 아래의 실례에서 설명되는데 이때 참조시간을 통과하는 통보문을 포함하는 객체의 구체례를 참조하기 위하여 클라스 Record 를 사용한다. 두 성원함수는 10 진수나 혹은 16 진수들중 어느 하나의 파라메터에 의한 통보문을 인쇄한다.

```
#ifndef  CLASS_RECORD_SPEC
#define  CLASS_RECORD_SPEC

class Record
{
public:
  Record( const char[ ] );                    // 이름을 기록
  void display_dec( const int ) const;  // 파라메터를 10 진수로 현시
  void display_hex( const int ) const;  // 파라메터를 16 진수로 현시
private:
  static const int SIZE = 40;
  char the_str[ SIZE ];                       // 통보문
};
#endif
```

```
#ifndef  CLASS_RECORD_IMP
```

```
#define  CLASS_RECORD_IMP
Record::Record( const char text[ ] )
{
  strcpy( the_str, text );                              // 통보문보관
}
void Record::display_dec( const int i ) const
{
   std::cout << the_str << “in decimal” << dec << i << “ \n”;
}
void Record::display_hex( const int i ) const
{
   std::cout << the_str << “ in hex ” << hex << i << “ \n”;
}
#endif
```

int 형파라메터를 가지고 void 형을 돌려 주는 클라스 Record 의 성원함수에 대한 성원지적자 p_fun 은 다음과 같이 선언된다.

```
void( Record::*p_fun )( const int ) const;
```

성원지적자 p_fun 은 필요한 명세부를 가지는 display_dec 나 display_hex 의 주소들중 어느 하나로 값주기될수 있다. 통보문 《Display point 1》을 가지는 클라스 Record 의 구체례는 다음의 선언에 의해 만들어 진다.

```
Record mes1 (“ Display point 1”);
```

성원함수 display_dec 의 주소는 다음의 선언으로 성원지적자 p_fun 으로 값주기될수 있다.

```
p_fun = &Record::display_dec;
```

주의: 콤파일러는 이 값주기에 대하여 형검사를 진행한다.

객체 mes1 에서 p_fun 으로 표시된 함수를 호출하자면 연산자 .*을 리용해야 한다. 이것은 함수와 클라스구체례사이에 정확한 결합이 이루어 질수 있도록 한다.

17.11.1 종합서술

다음의 프로그람은 함수 display_dec 와 display_hex 에 접근하기 위하여 연산자 .*과 ->*을 둘 다 리용한다.

```
int main( )
{
```

```
    Record mes1( "Display point 1" );
    Record *p_mes2 = new record( "Display point 2" );
    void ( Record::*p_fun)( const  int ) const;
    p_fun = &Record::display_dec;
    ( mes1.*p_fun )( 100 );                         // mes1 에서
    ( p_mes2 -> *p_fun )( 100 );                    // mes2 에서
    p_fun = &Record::display_hex;
    ( mes1.*p_fun )( 100 );                         // mes1 에서
    ( p_mes2 -> *p_fun )( 100 );                    // mes2 에서

    delete p_mes2;
    return 0;
}
```

주의: 객체를 직접 사용하지 않고 객체의 지적자를 사용할 때는 연산자 ->*이 .*과 같다.

실행결과는 다음과 같다.

```
Display point 1 in decimal 100
Display point 2 in decimal 100
Display point 1 in hex     64
Display point 2 in hex     64
```

17.12 지적자와 다형성

다형성은 일반적으로 지적자기구를 통하여 실현된다. 다음의 코드에서는 18.2 에 서술된 클라스 Abstract_Account, Account_IB, Account_IT 를 사용한다.

```
const int MAX_CUSTOMERS = 3;
Abstract_Account *customers[MAX_CUSTOMERS];
customers[0] = new Account   ( 100.00 );
customers[1] = new Account_IB( 10000.00 );
customers[2] = new Account_IB( 25000.00 );
```

주의: 어떤 기초클라스로부터 파생된 객체의 지적자는 기초클라스에 대한 형지적자의 기억위치로 값주기될수 있다.
만약 이것이 정확히 사용되지 못하면 오유가 발생할수 있다.

이것은 3 개의 서로 다른 구좌요소들을 설정한다.
은행구좌업무처리프로그람에서 다음의 코드는 매일마감에 계산서에 붙는 리자를 계산한다.

```
for( int i = 0;  i < MAX_CUSTOMERS;  i++ )
{
  customers[i] -> end+of _day( 365 );
}
```

리자는 새 회계주기의 첫날마감에 구좌에 첨가된다.

```
for( int i = 0;  i < MAX_CUSTOMERS;  i++ )
{
  customers[i] -> end+of _day( 1 );
}
```

다음의 코드를 리용하여 은행에서는 매 고객들의 구좌계산서를 인쇄한다.

```
for( int i = 0;  i < MAX_CUSTOMERS;  i++ )
{
    cout << “Customer” << i << “:” << “\n” << customers[i] -> statement( ) <<
 “\n” ;
}
```

만일 은행이 새 구좌형태를 도입한다고 해도 구좌의 업무처리에 대한 코드는 변경되지 않는다. 새로운 구좌를 도입하기 위해 요구되는 코드변경은 그 새 구좌에 대한 클라스에 교갑화되게 된다.

주의: 다형적호출이 진행될 때 객체와 함수호출사이의 결합은 반드시 실행시에 이루어져야 한다. 그러자면 함수가 가상적이여야 하며 지적자를 리용하든가 객체에 대한 참조를 리용하여 호출되여야 한다.

17.13 불투명형

동적기억기를 사용하면 객체에 대한 상태정보를 클라스명세부에서 삭제할수 있다. 클라스구축자는 객체에 기억기를 할당하며 해체자는 기억기를 체계에 돌려 준다. 실례로 클라스 Account 의 명세부는 다음과 같다.

```
#ifndef  CLASS_ ACCOUNT _SPEC
#define  CLASS_ ACCOUNT_SPEC

class Account
{
public:
  Account( const float = 0.00,  const float = 0.00  );
  ~Account( );
  float account_balance( ) const;                    // 잔고를 돌려 준다
```

```
  float withdraw(const float );                    // 구좌에서 출금
  void deposit( const float );                     // 구좌에 예금
  void set_min_balance( const float );             // 최소잔고를 설정한다
private:
  struct Account_data;                      // 림시선언
  Account_data*  the_storage;                      // 객체의 기억기에 대한 지적자
};

#endif
```

클라스 Account의 구축자와 해체자를 정의하면 다음과 같다.

```
#ifndef  CLASS_ ACCOUNT _IMP
#define  CLASS_ ACCOUNT_IMP

struct Account:: Account_data  {           // 객체의 물리적인 기억기
    float the_blance;
    float the_min_balance;
};

Account:: Account( const float money,  const float overdraft )
{
    the_storage = new Account_data[1];
    the_storage -> the_balance         = money;
    the_storage -> the_min_balance = overdraft ;
}

Account:: ~Account( );
{
    delete [ ] the_storage;
}
```

성원함수들의 실현부코드는 객체의 성원자료에 간접적으로 접근한다.

```
float  Account::account_balance( ) const
{
    return the_storage -> the_balance;
}

float Account::withdraw( const float money )
{
    if( the_storage -> the_balance = the_storege -> the_min_balance )
    {
        the+storage -> the_balance = the_storage -> the_balance - money;
        return money;
    } else {
        return 0;
```

```
    }
}

void Account::deposit( const float money )
{
    the_storage -> the_min_balance = the_balance + money;
}

void Account::set_min_balance( const float money )
{
    the_storage -> the_min_balance =  money;
}

#endif
```

알림

클라스 Account 의 구체례는 지적자들을 포함하기때문에 복사될수 없다. 객체와 같은것을 복사할 때에는 예견치 못했던 치명적인 결과가 초래될수 있다. 복사를 제한하는 리유와 그 대책에 대하여서는 깊은 복사와 얕은 복사를 서술한 23 장에서 보기로 한다.

17.13.1 클라스의 숨겨 진 기억기와 보이는 기억기

이 방법의 우점은 클라스의 의뢰자(client)가 그 클라스의 기억기구조가 변화되였을 때 자기 코드를 다시 번역할 필요가 없다는것이다. 의뢰자코드는 다만 새 실현부코드와 다시 련결되기만 하면 된다. 이것은 흔히 개선된 새로운 클라스서고가 쏘프트웨어공급자에 의해 제공될 때 생긴다. 이것은 그 서고의 대면부가 여전히 같다는것을 가정한다.

두 방법의 우결함은 다음과 같다.

기준	불투명형사용(숨겨 진 기억기)	표준형
콤파일효률	그 클라스만을 다시 콤파일하고 재련결을 실행하면 되므로 자원이 보다 적게 요구된다.	그 클라스를 사용하는 모든 단위들을 재콤파일해야 하므로 자원이 보다 많이 요구된다.
실행시 효률	동적기억기할당의 간접조작시간이 존재하므로 실행시 효률이 나빠 진다.	실행시 간접조작시간이 없다.
객체의 자료구성요소에로의 의뢰자접근	없다.	없다.
코드복잡성	기억기가 명시적으로 관리되여야 하므로 실현부에서 복잡성이 증대된다. 17 장을 보시오.	−

주의: 프로그람의 모든 단위를 다시 콤파일하여 련결하는것은 다시 련결만 하는데 비해 품이 많이 드는 작업이다.

17.14 클라스구성요소 *this

클라스에서 예약어 this 는 클라스의 현재구체례에 대한 지적자를 지적하는데 리용된다. 구체례에 접근하기 위하여 *this 가 리용된다. 15.2 의 초기실례에서 클라스 Money 에는 구좌량에 1 을 더하기 위한 증가연산자 ++가 정의되여 있었다. 다중정의된 증가연산자 ++는 *this 를 사용하여 클라스의 현재구체례를 결과로 내여 준다.

17.14.1 통보문과 메쏘드의 실현부

숨겨 진 첫번째 파라메터는 콤파일러에 의해 모든 비정적성원함수들에 삽입된다. 숨겨 진 파라메터 this 는 메쏘드가 호출하는 객체의 지적자이다. 실례로 객체 mike 에 다음의 통보문 deposit 를 보낸다.

```
Account mike;
Mike.deposit( 250.00 );
```

이것은 콤파일러에 의해 다음과 같이 실현된다.

```
deposit(& mike,250.00);
```

통보문에 의하여 호출된 메쏘드 deposit 는 다음과 같다.

```
void Account::deposit( float money )
{
  the_balance = the_balance + money;
}
```

이것을 콤파일러에 의해 다음과 같이 실현한다.

```
void deposit( Account *this, float money )
{
  this -> the_balance = this -> the_balance + money;
}
```

주의: 물론 콤파일시에 지정된 연산들의 유효성검사를 진행한다.
작성자는 this 를 명시적으로 사용하여 통보문을 받은 객체에 접근할수 있다.

17.15 자체평가

- 동적기억기할당과 변수의 표준선언에 의한 기억기할당사이의 차이는 무엇인가?
- 프로그람에서 동적기억기사용의 우점은 무엇인가?
- 연산자 ->가 중복되는가? 왜 그런가?
- 다음의 코드의 인쇄결과는 무엇인가?

```
char first[1], second[1];
*first = 'A'; *second = 'A';
cout << ( *first == *second ? “Equal”: “Not equal”);
cout << (  first ==  second ? “Equal”: “Not equal”);
```

- 다음의 의미는 무엇인가?
  - ✧ int *p_int;
  - ✧ int **p_P_int;
  - ✧ int *process(int)
- C++에서 연산자 []가 반드시 필요한가. 왜 그런가?
- 동적기억기사용은 흔히 어렵다고 한다. 이 말이 옳은가. 동적기억기접근을 보다 쉽게 하는 방법을 내놓을수 있는가?
- 지적자를 사용하는것이 왜 위험한가?
- 만일 객체가 어떤 기억기지적자를 가지고 있다면 왜 그것이 복사되지 않는가?
- 연산자 .*와 ->*의 목적은 무엇인가?

17.16 련습

다음의 클라스들을 작성하시오.

- 문자렬클라스

동적기억기를 리용하여 문자렬조작클라스를 작성하시오. 이 클라스는 그것을 사용하는 사용자가 다음과 같은것을 작성할수 있게 하는 메쏘드, 함수들을 가지고 있어야 한다.

```
String s1, s2;

s1.set(“Hello world”);  s2.set( “from brighton”;  s1.join_with( s2 );

cout << s1 << “\n”;
```

주의: 물론 자기가 만들어 놓은 기억기의 해제에 응당한 주의를 돌려야 한다.

알림

객체가 함수의 값에 의해 넘겨 지거나 함수의 결과로 돌려 질 때 암시적인 값주기(implied assignment)가 존재한다. 이 암시적인 값주기는 23 장에 서술되는 복사구축자에 의하여 조종된다. 만일 클라스 String 의 구체례가 함수의 값에 의해 넘겨 지거나 함수의 결과로 돌려 지면 복사구축자를 정의하지 않아도 지적자만은 복사되며 자료구조는 복사되지 않는다.

- 수클라스

임의의 정확도를 가지는 옹근수에 대한 연산을 진행하는 클라스를 작성하시오. 이러한 클라스는 실례로 다음의 처리를 할수 있어야 한다.

```
Number n1, n2;

n1.set( "12345678901234567890" );   n2.set( 12 );   n1.add( n2 );

cout << n1 << "\n" ;
```

18 다형성의 재고찰

이 장에서는 앞에서 취급한 다형성에 대하여 더 보기로 한다. 특히 추상클라스(abstract class)의 사용법과 실행시 형의 일치에 대하여 서술한다.

18.1 추상클라스

가상적인 성원함수들을 가지는 클라스를 정의할 때는 명세부(specification)만 정의하면 된다. 따라서 설계자는 클라스에 세부내용을 서술하지 않아도 되는 규약을 정의할수 있다. 이러한 형의 클라스를 추상클라스라고 한다. 이것은 다음의 책임들을 가지는 은행구좌클라스에 대한 규약에 의해 설명할수 있다.

메쏘드	책 임
account_balance	구좌의 잔고를 돌려 준다
deposit	구좌에 돈을 저금한다
end_of_day	날마감의 처리(그 날에 얻은 리자를 더하는것과 같은 처리)를 진행한다
set_min_balance	초과출금한계를 0.00으로 설정하여 초과출금을 하지 않는다
statement	구좌계산서를 표시하는 문자렬을 돌려 준다.
withdraw	자금이 충분하거나 초과출금이 허용될 때에만 구좌에서 돈을 내여 준다.

은행구좌클라스의 명세부는 다음과 같다.

```
#ifndef   CLASS_ABSTRACT_ACCOUNT_SPEC
#define   CLASS_ABSTRACT_ACCOUNT_SPEC
#include <string>
class Abstract_Account {
public:
virtual double account_balance( ) const  =  0;          // 잔고
virtual double withdraw( const double ) = 0;            // 출금
virtual void deposit( const double ) = 0;               // 저금
virtual void set_min_balance( const double ) = 0;       // 초과출금
virtual void end_of_day( const int ) = 0;               // 날의 마감
virtual std::string statement( ) const = 0;             // 계산서
virtual ~Abstract_Account( ) { };
};
#endif
```

주의: 추상클라스(혹은 가상함수의 기초정의를 포함하는 클라스)의 명세부에서 해체자는 가상적으로 지정된다. 만일 그렇게 하지 않으면 파생된 객체의 기억기가 해방될 때 다른 해체자가 호출될수 있다. 가상함수의 본체는 자기의 추상성을 나타내기 위해 '= 0'으로 지정된다.

추상클라스는 다음의 방법으로 리용될수 있다.

- 새로운 파생클라스의 기초클라스로서.
- 참조로 넘겨 지는 파라메터서술로서.
- 지적자서술로서.

따라서 아래의 선언들에서 첫번째것은 틀린 선언이며 두번째것이 옳은 선언이다.

```
Abstract_ Account   mine;                    // 콤파일할 때 오유를 발생
```

```
void   process (Abstract _Account&);         // 정확하다
Abstract_ Account  *mine;                    // 정확하다
```

18.1.1 일반구좌

그 어떤 리자도 없는 일반구좌를 클라스 Abstract_Account 에서 파생시켜 보자. 일반구좌클라스의 명세부는 다음과 같다.

```
#ifndef  CLASS_ACCOUNT_SPEC
#define  CLASS_ACCOUNT_SPEC
#include  "Abstract _account.h"

class Account  :  public Abstract_ Account   {
public:
   Account ( const double  =  0.0 );
  ~Account();
   double account_balance( ) const;              // 잔고
   double withdraw (const double );              // 출금
   void deposit (const double );                 // 저금
   void set _min_balance( const double );
                                                 // 초과출금
   void end_of_day(const  int);                  // 날의 마감
   std::string statement( ) const;               // 계산서
private:
    double the_balance;                          // 잔고열기
    double the_min_balance;                      // 초과출금
}:
#endif
```

주의: 함수 end_of_day 는 아무런 동작도 수행하지 않는다.

일반구좌클라스의 실현부는 다음과 같다.

```
#ifndef   CLASS_ACCOUNT_IMT
#define   CLASS_ACCOUNT_IMP
#include  <iostream>
#include  <strstream>
#include  < iopmanip>
Account : : Account ( const double  initial );
{
     the _balance         = initial;                          // 잔고열기
     the _min_balance = 0.00;                             // 초과출금을 허용하지 않는다
}
```

함수 account_balance, withdraw, deposit, set_min_balance 의 실현부는 다음과 같다.

```
double Account : : account_balance ( ) const
{
    return the_balance;
}
double Account : : withdraw (const double money)
{
     if   (the _balance_money >=  the_min_balance)
      {
          the _balance = the_balance - money;
          return money ;
       } else {
          return  0.00;
        }
}
void Account : : deposit (const double money)
{
     the_balance = the _balance + money;
}
void Account : :  set_min_balance (const double money)
{
     the_min_balance = money;
}
```

함수 end_of_day 의 본체는 비여 있으므로 클라스 Account 안에서 아무런 동작도 하지 않는다.

```
void  Account : : end _of_day (const  int  day)
{
          // 동작이 없다
}
```

주의: 기초클라스 Abstract_Account 에는 메쏘드 end_of_day 에 대하여 순수한 가상함수(pure virtual function)가 정의되여 있다. 따라서 클라스 Account 의 구체레를 창조하자면 메쏘드 end_of_day 를 구체적으로 정의하여야 한다.

함수 statement 는 구좌계산서를 표시하는 문자렬을 돌려 준다.

```
std :: string Account : : statement ( )  const
{
  const int MAX_BUF = 256;                // 본문기억령역의 최대크기
  char  buf[MAX_BUF];                      // 본문기억령역을 확보한다
  ostream text (buf, MAX_BUF);            // 본문은 std ::string 흐름이다
  text << setiosflags (ios ::fixed );           // 고정위치 x, y
  text  << setiosflags (ios :: showpoint );  // 모든 소수점아래자리를 보여 준다
  text << setprecision (2);                   // 소수점아래 2 자리
  text << "Balance of account is f " <<  account _balance ( );
  text <<  "\n"  << ' \o';
  return std : : string (buf);
}
```

해체자는 돈을 예금한 구좌가 닫겨 있다면 오유흐름에 경고를 보낸다.

```
Account : : ~Account ()
{
  if (account _balance () !=0.0 )
  {
      cerr << " Warning account contains  £" << account _balance ( );
      cerr << " \n ";
   }
}
# endif
```

주의: 매개의 순수가상함수(pure virtual function)는 클라스 Account안에서 다중정의(override)된다. 이것은 순수가상함수가 우연적으로 호출되는것을 방지한다. 콤파일러는 사용자가 순수가상함수를 포함하는 클라스구체레를 만들지 못하도록 한다.

18.2 파생된 리자산출구좌

리자산출구좌는 일반은행구좌가 가지고 있는것외에 다음의 책임들을 더 가진다.

메쏘드	책 임
prelude	이 구좌클라스들의 전체적인 리자률을 설정한다.
interest_day	미결제잔고에 관계되는 그날 리자를 계산한다.
credit_interest_day	회계주기마감이면 참을 돌려 준다.

여기에 이미 있던 메쏘드들의 책임을 첨부한다.

메쏘드	책 임
statement	이 구좌형에 대한 계산서를 인쇄한다.
set_min_balance	초과출금이 설정되는것을 방지한다.

이 리자산출구좌클라스 Account_IB의 명세부는 다음과 같다.

```
# ifndef CLASS_INTEREST_ACCOUNT_SPEC
# define CLASS_INTEREST_ACCOUNT_SPEC

class Account _IB : public Account {
public:
   Account _IB(const double = 0.0);              // 구좌
   ~Account_IB( );                               // 해체자
   static void  prelude(const double);
   std ::string statement( ) const;              // 구좌의 계산서
   void end_of_day (const int);                  // 날마감
   void set _min_balance(const double);          // 초과출금
protected:
   virtual  double  interest_today ( );          // 리자계산
   virtual bool credit_interest _day (const int);  // 회계주기마감
private:
   static double the_rate;                       // 리자률
   double the_accumulated_interest;              // 얻어 진 리자
};
#endif
```

주의: 성원함수 interest_today, credit_interest_day가 가상적으로 선언되는 리유는 18.3.3에서 서술한다.

클라스 Account_IB의 실현부는 다음과 같다.

```
#ifndef CLASS_INTEREST_ACCOUNT_IMP
#define CLASS_INTEREST_ACCOUNT_IMP
const double DAILY_RATE = 0.00026116;              // 년리자률 10%
double Account _IB : : the _rate;                  // 기억기를 선언
Account _IB :: Account _IB (const  double amount ): Account (amount)
{
   the_accumulated_interest = 0.00;
}
void Account _IB : : prelude(const double rate)
{
   the_rate = rate;
}
```

주의: 이 구좌에서 정적성원은 지불할수 있는 리자률을 기록하기 위해 사용된다.

해체자는 어떤 지정된 날(즉 0 인 날)에 대한 날마감처리를 호출한다. 이 지정된 날은 그 구좌에 요구되는 어떤 닫기동작을 불러 낸다.
Account_IB::~Account_IB()

```
Account _IB : :~Account_IB( )
{
    end_of_day(0);                                  // 리자를 붙인다
}
```

함수 end_of_day는 매일 리자계산에 호출되며 구좌에 붙어 난 리자를 저축한다. 리자기입날에 붙어 난 리자가 구좌에 기입된다.

```
void Account_IB ::end_of_day (const int day)
{
    if ( day !=0 )
    the_accumulated _interest += interest_today( );        // 보통날
     if ( credit_interest_day(day) )                        // 마감날
     {
          deposit( the_accumulated_interest );              // 구좌에 저금
          the_accumulated_interest = 0.0;
      }
}
```

주의: 0인 날에는 지정된 동작이 진행된다. 이 날에는 구좌에 대한 닫기동작을 불러 낸다.

성원함수 interest_today와 credit_interest_day에서 다형적호출(polymorphic call)이 이루어 진다. 함수본체의 실현부안에서 행

```
the_accumulated_interest += interest_today( );
```

이

```
the_accumulated_interest += this->interest_today( );
```

으로 실현된다는것을 기억하시오.
함수 interest_today는 그날의 미결제잔고와 관련하여 얻어 진 리자량을 돌려 준다.

```
double Account_IB::interest_today( )
{
   return account_balance( ) *the_rate;
}
```

계산주기의 마감에 리자가 구좌에 기입되며 함수 credit_interest_day는 이날에 참을 돌려 준다.

```
bool Account_IB :: credit_interest_day (const int day)
{
    return  day ==1 || day ==0;             // 리자기입날
}
```

성원함수 set_min_balance는 기초클라스안의 메쏘드를 다중정의하여 이 메쏘드가 직접 호출되는것을 방지한다.

```
void Account_IB ::set_min_balance(const double)
{
    return;          // 동작이 없다
}
```

성원함수 statement는 클라스 Account안의 statememt를 호출하기 위해 유효범위해결연산자 ::를 사용한다.

```
std::string Account _IB::statement( ) const
{
    return std::string( "Interest bearing Account : ") + " \n " + Account :: statement( );
}
#endif
```

주의: 첫번째 문자렬은 첫번째와 두번째 문자렬이 련결되도록 클라스 string의 구체례로 변환되여야 한다.

18.3 고리자구좌의 파생

리자률(interest rate)들이 계층구조를 이룬 리자산출구좌는 리자산출은행구좌(interest bearing bank account)가 가지고 있는것외에 다음의 책임을 더 가진다.

메쏘드	책 임
prelude	전체 리자률을 설정한다.

여기에 이미 있던 메쏘드들의 책임을 첨부한다.

메쏘드	책 임
interest_today	미결제잔고로 인한 리자를 계산한다.
statement	구좌계산서를 표시하는 문자렬을 돌려 준다.

클라스 Account_T 의 명세부는 다음과 같다.

```
#ifndef  CLASS_SPECIAL_INTEREST_ACCOUNT_SPEC
#define  CLASS_SPECIAL_INTEREST_ACCOUNT_SPEC
class Account_IT : public Account_IB {
public:
   Account_IT(const double = 0.0);
   static void prelude (const double, const double, const double);
   std:: string statement ( ) const;          // 구좌계산서
protected:
  double interest_today ( );                  // 그날리자를 계산
private:
  static double the_interest_rate1;        // 리자률 1
  static double the_interest_rate2;        // 리자률 2
  static double the_interest_rate3;        // 리자률 3
};
#endif
```

클라스 Account_IB 의 실현부는 다음과 같다.

```
#ifndef CLASS_SPECIAL_INTEREST_ACCOUNT_IMP
#difine CLASS_SPECIAL_INTEREST_ACCOUNT_IMP

const double DAILY_RATE_R1 = 0.00026116;   // 년리자률 10%
const double DAILY_RATE_R2 = 0.00028596    // 년리자률 11%
const double DAILY_RATE_R3 = 0.00031054;   // 년리자률 12%

double Account_It::the_interest_rate1;               // 리자률 1
double Account_It::the_interest_rate2;               // 리자률 2
double Account_It::the_interest_rate3;               // 리자률 3
```

구축자는 클라스 Account_IB 의 구축자를 호출한다.

```
Account_IT::Account_IT(const double  amount) : Account_IB( amount )
{
}
```

정적성원함수 prelude는 클라스안에서 객체들이 사용하는 계층적인 리자률들을 설정한다.

```
void Account_IT::prelude (
    const double r1=DAILY_RATE_R1,  const double r2=DAILY_RATE_R2,
    const double r3=DAILY_RATE_R3 )
{
```

```
    the_interest_rate1=r1;
    the_interest_rate2=r2;
    the_interest_rate3=r3;
}
```

성원함수 interest_today 와 statement 의 실현부는 다음과 같다.

```
double Account_IT :: interest_today( )
{
    double money = account_balance( );
    if ( money < 10000.00)
        return  money * the_interest_rate1;   // 리자범위 1
    else if ( money < 25000.00 )
        return  money * the_interest_rate2;   // 리자범위 2
    else
        return  money * the_interest_rate3;   // 리자범위 3
}

std::string Account_IT::statement( ) const
{
    return std::string( "Special Interest bearing Account:" ) +  " \n " +
            AccountA::statement();
}

#endif
```

18.3.1 종합서술

다음의 함수는 우에서 서술한 3 가지 형태의 구좌클라스를 사용한다. 여기서 머리부파일들의 포함과 앞에서 설명한 리자률들의 설정은 생략한다.

```
// 해당머리부파일들을 포함
// 구좌들의 리자률을 설정

void process( )
{
    const int MAX_CUSTOMERS = 7;
    Abstract_Account *customers[MAX_CUSTOMERS];
    int i;

    customers[0] = new Account       (10000.00);
    customers[1] = new Account_IB  (10000.00-0.01);
    customers[2] = new Account_IB  (10000.00);
    customers[3] = new Account_IT  (10000.00-0.01);
    customers[4] = new Account_IT  (10000.00);
    customers[5] = new Account_IT  (25000.00-0.01);
```

```
    customers[6] = new Account_IT  (25000.00);
// 은행은 매일 마감에 매 구좌에 기입되는 리자를 계산한다
   for (i = 0; i < MAX_CUSTOMERS; i ++)
   {
       customers[i] → end_of_day(365);
   }
// 매 계산주기 마감에 리자가 매 구좌에 기입된다
   for (i = 0; i < MAX_CUSTOMERS; i ++)
   {
      customers[i]→ end_of_day(1);
   }
    for ( i = 0; i < MAX_CUSTOMERS; i ++)
    {
        std::cout << "Customer" <<i<< ": " << "\n"<< customers[i] →
statement( )<< "\n";
     }
    for ( i = 0; i < MAX_CUSTOMERS; i ++)
    {
        delete customers[i];
     }
}
```

실행결과는 다음과 같다.

```
Customer 0:
Balance of account is £ 10000.00
Customer 1:
Interest  bearing Account:
Balance of account is £ 10005.21
Customer 2:
Interest  bearing Account:
Balance of account is £ 10005.22
Customer 3:
Special Interest  bearing Account:
Balance of account is £ 10005.21
Customer 4:
Special Interest  bearing Account:
Balance of account is £ 10005.72
```

```
Customer5:
Special Interest  bearing Account:
Balance of account is ￡ 25014.29

Customer6:
Special Interest  bearing Account:
Balance of account is ￡ 25015.53
```

다음의 경고문은 구좌가 닫길 때 그속에 여전히 돈이 들어 있는것과 관련하여 발생된다.

```
Warning account contains ￡ 10000
Warning account contains ￡ 10005.2
Warning account contains ￡ 10005.2
Warning account contains ￡ 10005.2
Warning account contains ￡ 10005.7
Warning account contains ￡ 25014.3
Warning account contains ￡ 25015.5
```

18.3.2 호출된 성원함수

다음의 표는 매 클라스에 의해서 제공된 성원함수들을 보여 준다. 클라스구체례에서 호출된 성원함수가 그 클라스에 정의되여 있지 않으면 클라스계층(class hierarchy) Abstract_ Account >>Account_>>Account_IB>>Account_IT 의 마지막기초클라스에서 정의된다. 이것은 클라스 Account 가 반드시 매 가상함수에 대한 코드를 가져야 한다는것을 말해 준다. 즉 클라스 Account 안에 가상함수가 정의되여 있지 않으면 콤파일러는 실현부가 순수한 가상함수를 제공하지 못한다는 오유통보문을 내보낸다.

Abstract_Account	**Account**	**Account_IB**	**Account_IT**
account_balance △	account_balance	➡	➡
		interest_today △	interest_today
		credit_interest_day △	➡
deposit △	deposit	➡	➡
end_of_day △	end_of_day	end_of_day	➡
		prelude	prelude
set_min_balance △	set_min_balance	set_min_balance	➡
statement △	statement	statement	statement
withdraw △	withdraw	➡	➡
	Constructor	Constructor	Constructor
~Destructor △		~Destructor	

주의: ➡는 계승메쏘드를 표시한다.
Δ은 가상함수정의를 표시한다.
추상클라스는 구축자를 가지지 못하지만 가상해체자를 가진다. 이것은 어떤 계승클라스로부터 만들어 진 객체가 정확한 해체자를 호출할수 있다는것을 말해 준다. 만일 구축자나 해체자가 클라스안에 정의되여 있지 않으면 그의 실현부는 콤파일러에 의해 기정으로 제공된다.

18.3.3 재발송

클라스 Account_IB 안에 있는 메쏘드 interest_today 의 실현부는 다음과 같다.

```
void Account_IB::end_of_day(const int day)
{
  if ( day !=0 )
  the_accumulated_interest += interest_today ( );    // 보통날
  if ( credit_interest_day(day) )                      // 마감날
  {
    deposit ( the_accumulated_interest );              // 구좌에 저금
    the_accumulated_interest = 0.0;
  }
}
```

통보문 interest_today 와 credit_interest_day 가 발송될 때 호출되는 실제메쏘드(actual method)는 통보문 end_of_day 를 받는 객체의 형에 의존한다.

```
Abstract_Account * object_ib = new Account_IB( 25000.00 );
Abstract_Account * object_it = new  Account_IT( 25000.00 );
```

실례로 우의 선언들에서 통보문 end_of_day 가 매 객체들에 보내질 때 호출되는 메쏘드들의 결과는 다음과 같다.

object_ib->end_of_day();	object_it->end_of_day();
Account_IB::end_of_day() Account_IB::interest_today() Account_IB::credit_interest_day() Account ::deposit(⋯)	Account_IB::end_of_day() Account_IT::interest_today() Account_IB::credit_interest_day() Account ::deposit(⋯)

보는바와 같이 실제메쏘드 interest_today 의 호출은 통보문 end_of_day 를 받는 객체의 형에 의존한다. 클라스 Account_IB 작성자는 메쏘드 interest_today 가 가상적으로 선언되여 계승클라스의 새로운 메쏘드에 의해 다중정의될수 있으므로 이 메쏘드의 어느것이 호출되는가에 대해서는 알수 없다.

알림

이런 방법을 써서 문서화되지 않으면 end_of_day에 대한 우의 실현부를 믿기가 어렵다. 클라스 Account_IB의 새로운 실현부는 interest_today의 다형적호출을 포함하지 않는 방법에 의하여 메쏘드 end_of_day를 실현할수 있다.

18.4 실행시 typeid

이종집합(heterogeneous collection)에서 객체들이 기초클라스의 지적자로 서술될 때 그 집합의 사용자는 객체에 대한 본래의 형정보를 잃게 된다. 그러나 C++는 모든 객체들에 형일치(type identity)를 제공한다. 예약어 typeid 는 객체의 형을 돌려 주는데 사용된다. 실례로 손님들의 이종집합에 대한 모든 리자산출구좌계산서를 출력하는 코드는 다음과 같다.

```
int main()
{
    const int   NO = 4;
    Account  *customers [NO];
    customers[0] = new Account      ( 100.00 );
    customers[1] = new Account_IB( 200.00 );
    customers[2] = new Account      ( 300.00 );
    customers[3] = new Account_IB( 400.00 );
    std::cout << "List of interest bearing accounts:"<< " \ n "
    for ( int i = 0; i < NO; i ++)
    {
       if  ( typeid(*customers[i]) = = typeid(Account_IB) )
       {
            std::cout<< "Customers account # " < i << " " <<
                                                customers[i] -> statement();
        }
     }
    for ( int r = 0; r < NO; r ++)
    {
        delete  customers[r];
    }
    return 0;
}
```

주의: Account_IB 형의 내부표시를 돌려 주기 위해 typeid(Account_IB)를 사용한다.

실행결과는 다음과 같다.

```
List of interest bearing accounts:
Customers account #1 Interest bearing Account:
Balance of account is £ 200.00
Customers account #3 Interest bearing Account:
Balance of account is £ 400.00
```

18.4.1 클라스 type_info

실행시 typeid 는 프로그람작성자가 실행시(run_time)에 객체에 질문하고 그의 형을 결정하는 능력을 가질수 있게 한다. 우에서 본바와 같이 이 기능은 이종집합에서 객체의 실제적인 형을 결정할 때 효과적이다. typeid 는 type_info 형객체에 참조를 돌려준다. 클라스 type_info 의 공개(public)명세부는 다음과 같다.

```
class type_info {
public:
   virtual ~type_info( );
   bool operator = = (const type_info & rhs) const;
   bool operator != (const type_info & rhs) const;
   bool before (const type_info & rhs) const;
   const char* name( ) const;
};
```

클라스 type_info 는 머리부파일 <typeinfo>에 정의되여 있다.

18.5 내리변환

내리변환(downcasting)은 기초클라스구체례를 파생클라스구체례로 변환하는것이다. 이 변환은 보통 기본형객체가 파생형객체로 변환되게 하는 추가정보를 기본형객체에 첨가해야 하므로 불가능하다. 그러나 어떤 프로그람에서는 파생형구체례의 지적자를 기본형구체례의 지적자로 서술할수 있다. 이것은 흔히 이종집합을 창조할 때 이루어진다. 이종집합의 자료성원들은 서로 다른 형들로 구성되여 있지만 각기 집합기본형의 지적자로 정의된다.

물론 기본형으로부터 파생형에로의 변환은 가능해야 한다. 실례로 이종배렬 customers 안에 있는 리자산출구좌들을 배렬 better_accs 에 복사하는 코드는 다음과 같다.

```
Account_IB   better_accs[NO];
int              number_better_accs = 0;

for ( i = 0; i < NO; i ++ )
{
   Account_IB  *p_acc = dynamic_cast<Account_IB*> ( customers[i] );
```

```
    if ( p_acc != NULL )
   {
       better_accs [number_better_accs++] = *p_acc;
    }
}
```

주의: 만일 강제형변환을 할수 없다면 연산자 dynamic_cast 는 NULL 을 돌려 준다.

연산자 dynamic_cast 는 실행시 형검사에 의하여 강제형변환이 진행되도록 한다. dynamic_cast 연산자의 형식을 그림 18-1 에 보여 준다.

객체를 강제형변환 본래의 객체

dynamic_cast <Account_IB*>(customers[i])

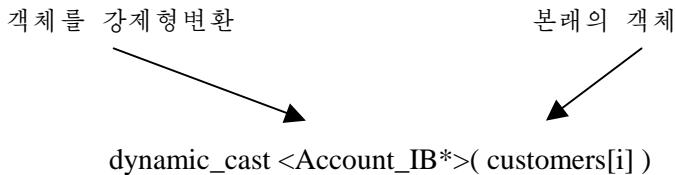



그림 18-1. 연산자 dynamic_cast 의 구성요소들

이것과 다른 강제형변환요소들에 대해서는 19.6 에서 설명한다.

18.6 자체평가

- 추상클라스에서는 왜 기억기선언을 할수 없는가?
- 실현자(implementor)는 왜 추상클라스명세부를 제공해야 하는가?
- 그 객체에 해당한 클라스를 어떻게 찾을수 있는가?
- 실행시 형의 일치가 왜 필요한가?
- 프로그람작성자가 내리변환을 사용하는것이 왜 필요한가?

18.7 련 습

- 은행

  각이한 형태의 구좌에 대하여 일반적인 업무처리를 진행하도록 하는 은행체계 골격도를 만드시오. 이 체계의 목적은 그 체계에 대한 코드를 변화시키지 않고도 새로운 형태의 구좌가 그이후의 단계에 추가될수 있도록 하는데 있다.

- 사람과 콤퓨터사이에 진행하는 C4 유희

  지적자와 다형성을 리용하여 12 장에서 보여 준 C4 유희를 다시 실현하시오. 클라스 Player 의 메쏘드 get_move 는 배렬 contestants 가 사람뿐아니라 콤퓨터선수까지 포함하도록 가상적으로 만들어 질수 있다. 경기자들에 대한 선언은 다음과 같이 될수 있다.

```
Board  c_4;

Player*  contestant[] = {
   new Player( Counter (Counter::BLACK) ),
   new Computer_Player ( Counter (Counter::WHITE), c_4),
};
```

메쏘드 get_move 는 기초클라스 Basic_player 안에서 가상적으로 만들어 진다. Computer_player 에 대한 구축자는 쓰인 말을 기억시킨것외에도 유희판의 지적자도 기억시킨다. 따라서 메쏘드 get_move 는 유희판에 문의할수 있다. 메쏘드 play 의 동작을 처리하는 코드는 다음과 같다.

```
move = contestant [no] -> get_move (terminal);
while ( !c_4.move_ok_for_column(move) )
 {
   move = contestant [no] -> get_move(termina,true);
 }
 c_4. drop_in_column(move, contestant[no] -> counter_is() );
```

19 선언과 강제형변환

이 장에서는 C++에서 파생형들에 대한 기억기를 선언하는 방법을 보기로 한다. 기억기를 선언할 때 암시적으로 할당되지 않았던 자료구조부분의 기억기를 명시적으로 할당하는것이 중요하다. 이밖에도 명시적인 강제형변환기구에 대해서도 서술한다.

19.1 파생형의 기억기선언

C++에서는 단일형이 모여 복합형을 이루는 구조체들을 직접 선언할수 있다. 그러나 파생형에 대한 직접선언에서는 대부분 기억기의 일부만이 암시적으로 할당되고 그 나머지부분은 모두 명시적으로 할당, 결합되여 전체 기억기에 그 구조체가 하나하나 서술되여야 한다.

```
int  *table;
```

우의 실례는 int 형지적자를 가질수 있는 기억위치 table 의 선언이다. 그 기억위치 table 의 내용은 정의되여 있지 않으며 이에 대한 명시적인 값주기가 옹근수를 가질수 있는 기억세포에 대한 지적자를 제공할수 있도록 table 이 만들어 져야 한다.

이러한 선언에서는 연산자 특히 연산자 [], (), *의 우선권이 중요하다.

19.1.1 []와 (), *의 우선순위

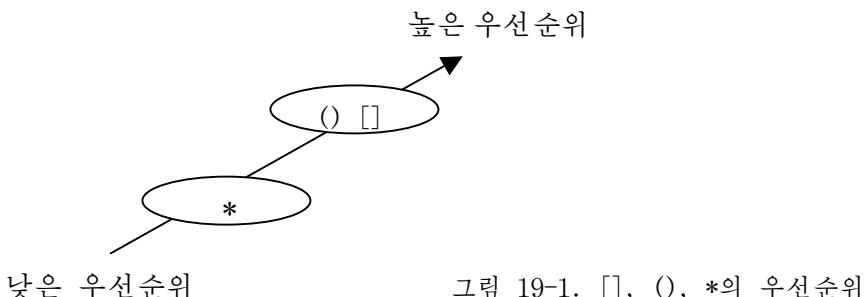
이것을 그림 19-1 에서 설명한다.

높은 우선순위

() []

*

낮은 우선순위

그림 19-1. [], (), *의 우선순위

개요

연산자	선언변수와 결합될 때	설명	우선순위
[]	왼쪽에서 오른쪽으로	배렬선언	높다(()와 같다).
()	왼쪽에서 오른쪽으로	함수선언	높다([]와 같다).
*	오른쪽에서 왼쪽으로	간접	낮다.

```
int   *table[3];
```

이 명령문은 옹근수형항목들에 대한 세개의 지적자배렬을 선언하는데 이때 그 배렬에 대한 기억기만이 할당된다. 3개의 옹근수를 보유하는 기억기는 따로따로 할당되여야 한다.

이 선언은 다음과 같이 읽는다.

```
int   *table[3];        3개 배렬
int   *table[3];        3개 지적자들의 배렬
int   *table[3];        함수형항목들에 대한 3개 지적자들의 배렬
```

주의: 선언다음에 인차 자료구조체에 필요한 추가기억기를 할당하는것이 좋다.

```
int   ( *table )[3];
```

이것은 3개 옹근수형항목배렬에 대한 지적자를 선언한다. 이때는 이 구조체의 지적자를 가지는 기억기만이 할당된다.

실례로 앞에서 본 연산자우선순위규칙을 리용하여 이 선언을 다음과 같이 읽을수 있다.

```
int  (*table)[3];   지적자
int  (*table)[3];   3개 항목배렬에 대한 지적자
int  (*table)[3];   3개 옹근수형항목배렬에 대한 지적자
```

주의: []은 *보다 우선순위가 더 높다.

19.2 구조체할당

C++에서 선언은 2가지 속성들을 가진다.

- 자료구조체의 전체 또는 부분을 위한 기억기의 창조.
- 자료구조요소들에 대한 접근방법을 콤파일러에 알려 준다.

매 선언에 대한 다음의 실례들은 기억기의 서술을 그림으로 보여 준다. 실선으로 표시된 구역은 암시적으로 할당된 기억기이고 점선으로 표시된 구역은 후에 명시적으로 할당되는 기억기이다.

선 언	자료구조
int value;	
int value; int 항목에 대한 지적자	
int value [3]; 3 개의 int 형항목배렬	
int *value[3]; int 형항목에 대한 3 개의 지적자배렬	
int (*value) [3]; 3 개의 int 형항목배렬에 대한 지적자	
char *(*argv)[ ]; 어떤 문자에 대한 열린 배렬지적자들에 대한 지적자	

19.3 함수원형

앞에서 본바와 같이 모든 함수들은 사용되기전에 함수원형명령문에 의하여 선언되여야 한다. 그러나 다음의 실례들에서 돌려 지는 항목은 상대적으로 단순하다.

함수원형을 보기로 하자.

```
char *return_str();
```

함수 return_str 는 char 형항목의 지적자를 돌려 주는 파라메터가 없는 함수이다. 읽는 방법은 다음과 같다.

char *return_str(); 파라메터가 없는 함수.
char *return_str(); char 형항목의 지적자를 돌려 주는 파라메터가 없는 함수.

주의: 함수에 주어 진 이름이 return_str 이므로 그 지적자는 C++문자렬로 이루어 진 문자배렬의 첫번째 문자를 가리키게 된다.

좀 더 복잡한 선언을 보기로 하자.

```
char * ( *fun ( int ) )  ( );
```

int 형파라메터를 가지는 함수의 지적자를 돌려 주는 파라메터가 없는 함수는 C++ 문자렬에 대한 지적자를 돌려 준다.

이것을 읽는 방법은 다음과 같다.

char * (<u>*fun(**int**)</u>) ( );	int 형파라메터를 가지는 함수지적자.
char * <u>(*fun(**int**)) ( )</u>;	int 형파라메터를 가지는 함수의 지적자를 돌려 주는 파라메터가 없는 함수.
<u>**char** * (*fun(**int**)) ( )</u>;	int 형파라메터를 가지는 함수의 지적자를 돌려 주는 파라메터가 없는 함수는 C++문자렬에 대한 지적자를 돌려 준다.

이것은 예약어 typedef 를 사용하여 더 쉽게 표현할수 있다.

```
typedef char*     C_string;               // C++문자렬
typedef C_string  F_C_string(int);        // C_string 을 돌려 주는 함수

F_C_string*       fun( );                 // F_C_string 지적자를 돌려 주는 함수
```

19.3.1 형식파라메터선언

기억기를 선언하는것과 마찬가지로 함수의 파라메터들도 선언되여야 한다.

19.4 공용체

공용체(union)는 여러개의 자료항목들이 같은 물리적기억기를 공유하게 된다는것을 가리킨다. 공용체는 항목들이 기억되지 않고 다같이 사용될 때 기억기를 매우 효과적으로 리용할수 있게 한다. 공용체사용에서의 주의점은 주로 새로운 자료항목이 삽입될 때 공용체안에 이미 기억되여 있던 자료항목이 지워 지고 다른 형으로 기억된 자료항목이 추출된다는것이다. 실례로 int, char, float 를 가지는 공용체는 int 형값을 기억하는데 이 값은 float 형처럼 추출될수 있다.

주의: 때때로 자료항목을 다른 형으로 추출해 내는것이 유용할수도 있다.

다음의 프로그램에서 공용체는 char 또는 int, float 를 가지는 Store 형을 정의한다.

```
int main()
{
    union Store
    {
```

```
        char    as_a_char;              // 이 세가지 형들은
        int     as_an_int;              // 같은 기억공간을
        float   as_a_float;             // 차지한다
    };
    Store  item;
    item.as_a_char = ‘A’;               // 문자를 가지는 항목을 사용
    item.as_an_int = 42;                // 옹근수를 가지는 항목을 사용
    item.as_a_float = 3.14;             // 류점수를 가지는 항목을 사용
    cout << “int=” << item.as_an_int << “\n”;
    return  0;
}
```

실행결과는 다음과 같다.

```
int = -2621
```

이것은 기대하던 결과가 아니다.

19.5 비트마당

struct 는 int 형비트들의 정확한 개수를 지적하는 비트마당수식자를 가질수 있다. 이것은 콤파일러실현자에 의해서 이루어 진다. 다음의 프로그람에서는 여러가지 상태 기발들을 표시하는 기계단어가 들어 있는 Flags 라는 이름의 구조체를 선언한다.

```
int main( )
{
    struct Flags
    {
        unsigned  int  status : 3;
        unsigned  int  size   : 6;
        unsigned  int  ok     : 1;
    };
    Flags   operation;
    operation.status = 3;
    operation.size   = 5;
    operation.ok     = 1;
    std :: cout << “status :” << ( int ) operation.status << “ \n ”;
    std :: cout << “size   :” << ( int ) operation.size  << “ \n ”;
    std :: cout << “ ok    :” << ( int ) operation.ok << “ \n ”;
    std :: cout << “word :” << oct << (reinterpret_cast<int>(&operation)) <<
 “ \n ”;
```

```
    return 0;
}
```

주의: 강제형변환 reinterpret_cast 는 Flags 형지적자를 int 형지적자로 변환한다.

실행결과는 다음과 같다.

```
status : 3
size   : 5
ok     :1
word  :1053
```

주의: unsigned int 형은 부호확장이 없는 int 형이다.

19.6 강제형변환

C++는 어떤 형의 객체를 다른 형의 객체로 변환하는 여러가지 명시적인 방법들을 제공한다. 이러한 명시적인 강제형변환기구들은 다음과 같다.

- const_cast <T> (arg)
- dynamic_cast <T> (arg)
- reinterpret_cast <T> (arg)
- static_cast <T> (arg)

여기서 arg 는 T 형으로 강제형변환되게 되는 식이다.

- const_cast <T> (arg)
  상수강제형변환은 객체에 const 나 volatile 변경자(modifier)를 추가하거나 삭제하는데 리용된다. 다음의 코드는 const 변경자의 추가와 삭제이다.

```
const char  *const_name = ” mike” ;

char  name = const_cast<char*>( const_name );

const char  const_name2 = const_cast < const char*>( name );
```

- dynamic_cast <T> (arg)
  동적강제형변환은 객체의 지적자나 참조를 다른 객체의 지적자나 참조로 변환하는데 리용된다. 이 변환의 사용에서 중요한것은 기초클라스와 파생클라스사이의 변환이다. 강제형변환의 가능성은 실행시에 검증된다. 지적자강제형변환이 실패되면 0 이 돌려 지고 참조강제형변환이 실패되면 례외 bad_cast 가 내보내진다.
  다음의 코드는 이종집합으로부터 파생클라스의 구체례를 회복한다.

```
Base   *collection[2];
collection[0] = new Derived[1];
collection[1] = new Derived[1]
```

```
Derived   *d = dynamic_cast<Derived*>( collection[0] );
if ( d == 0)
{
    // 실패
}
```

- reinterpret_cast <T> (arg)
  이 변환은 일반적으로 어떤 형의 지적자를 다른 형의 지적자로 강제형변환하는데 리용된다. 이것은 int 형을 지적자로 변환하는것과 그의 반대과정을 포함한다. 변환에서 T 는 지적자, 참조, 수값형, 함수지적자나 성원지적자이여야 한다.
  다음의 코드는 문자상수를 char 형, int 형지적자로 변환한다.

```
char   *p_ch = reinterpret_cast<char*>( 0x0000FA00  );
int     *p_int = reinterpret_cast<int>( p_ch );
```

- static_cast<T>(arg)
  이 변환은 어떤 식을 콤파일러가 현재의 변환기구에 의하여 수행할수 있는 형으로 강제형변환하는데 사용된다.
  실례로 다음의 코드는 char 형을 int 형으로, 파생클라스를 기초클라스로 변환한다.

```
char   c = ‘M’;
int   as_int = static_cast<int>( c );

Derived   d;
Base       b = static_cast<Base>( d );
```

19.7 자체평가

- 다음의 두 선언의 차이는 무엇인가?

```
int    value[3];
int   *value[3];
```

- 문자객체에 대한 3 개의 열린 배렬지적자들에 대한 지적자를 표현하는 객체 data 를 선언하자면 어떻게 해야 하는가?
- 공용체를 사용하면 왜 위험한가?
- 언제 dynamic_cast 를 사용하는가?

20 용기클라스

이 장에서는 벡토르의 실현부를 안전하게 작성하기 위한 연산자 []의 다중정의방법에 대하여 서술한다. 이것은 STL(Standard Template Library : 표준본보기서고)본보기클라스 Vector 의 대면부에서 부분적으로 모형화된다. STL 서고에 대해서는 24 장과 25 장에서 서술한다.

20.1 소 개

C++배렬에서 중요한것은 실행시에 배렬경계가 검사되지 않는것이다. 물론 이것은 실행시 어떤 심한 오유를 낼수 있는데 예상치 않던 고장까지도 일으킬수 있다. 더욱 나쁜것은 오유가 발견되지 못하고 틀린 결과가 출력되는것이다.

C++에서 첨수를 사용한 배렬의 접근은 지적자와 주소를 사용하는 간단한 방법이다. 실례로 다음의 배렬선언을 보기로 하자.

```
int  vec[10];
```

첨수식

```
int  res = vec[5];
```

는 아래와 같이 쓸수 있다.

```
int  res = *( & vec[0] + 5 );
```

지적자에 값을 더하거나 덜 때 C++는 C 와 마찬가지로 그 값에 지적된 기억기의 크기를 곱한다.

식	콤파일러에 의한 해석
*(& vec[0]+5)	*(& vec[0]+5 * sizeof(int))

이러한 수법은 콤파일러가 배렬의 표준적인 첨수를 평가할 때도 사용된다. 그 식의 항목대부분이 콤파일시에 알려 지므로 코드생성은 매우 간단하다.

20.1.1 풀이

C++에서는 연산자 []를 재정의할수 있으므로 발견되지 않은 실행시 오유들에 의한 위험을 피할수 있다. 이렇게 벡토르를 안전하게 실현하는 클라스를 아래에 보여준다. 이 클라스를 실현하는데서 그의 구체례가 2 차원배렬을 안전하게 만드는데 리용될수 있도록 하는 방법들이 제공된다.

20.2 안전한 벡토르

안전한 벡토르(safe vector)의 책임은 다음과 같다.

메쏘드/구축자	책 임
( )	경계검사없이 요소에 접근한다.
[ ]	경계검사를 하면서 요소에 접근한다.
back	벡토르의 마지막요소를 돌려 준다.
front	벡토르의 첫번째 요소를 돌려 준다.
pop_back	끝요소를 삭제하는것으로 벡토르의 크기를 변경시킨다.
push_back	끝에 요소를 추가하는것으로 벡토르크기를 변경시킨다.
set_def_size	벡토르를 만들기 위한 기정크기를 설정한다.
size	벡토르의 길이를 돌려 준다.
Vector	안전한 벡토르를 생성한다.

주의: 연산자 []는 저준위첨수연산자, 연산자 ()는 함수호출연산자이다.

20.2.1 클라스 Vector 구체례의 사용에서의 제한

Vector 구체례는 복사될수 없는 제한을 가지고 있다. 이 제한은 콤파일러에 의한 것이 아니다. 그 리유에 대해서는 제한을 실시하는 기구와 지적자를 가지는 객체들이 복사될수 있게 하는 처리를 설명하는 23장에서 구체적으로 서술한다.

20.2.2 클라스 Vector 의 명세부

클라스 Vector 의 명세부는 다음과 같다.

```
template <class Type>
class Vector {
public:
   typedef  Type  Value_type;                      // 볼수 있게 한다
   explicit Vector ( size_t = 0 );                 // 구축자
   ~Vector ( );                                    // 해체자
   Type&  operator ( ) ( size_t );                 // 검사하지 않는 접근
   const Type&  operator ( ) ( size_t )  cosnt;    // const 를 검사하지 않는 접근
   Type& operator [] ( size_t );                   // 검사하는 접근
   const Type& operator [] ( size_t )  cosnt;      // const 를 검사하는 접근
   size_t size ( ) const;                          // 길이
   void push_back ( const Type& );                 // 끝에 요소를 추가
   void pop_back ( );                              // 끝에서 요소를 삭제
   inline const Type&  front ( ) const;            // 앞요소
```

```
   inline Type& front ( );                                 // 앞요소
   inline const Type& back ( ) const;                      // 뒤요소
   inline Type& back ( );                                  // 뒤요소
   static void set_def_size ( size_t );
protected:
  void fail (const char[],  size_t ) const;
  Type* new_vec (size_t);
private:
  static const int  INC_SIZE =10;                          // 확장크기
  Type*  the_p_item;                                       // 배렬지적자를 기억
  size_t   the_vec_size;                                   // 배렬안의 요소들
  size_t   the_vec_actual_size;                            // 배렬의 실지크기
  static int  the_def_size;                                // 기정크기
};
```

주의: 다중정의연산자 []는 클라스의 고정구체례(const instance)를, ()는 클라스의 가변구체례(non const instance)를 제공한다.
역시 메쏘드 front는 클라스의 고정구체례를, back는 가변구체례를 제공한다.

옹근수벡토르를 다음과 같이 선언할수 있다.

```
Vector<int>   numbers;
Vector<int>   many_numbers( 25 );
```

이것은 벡토르가 C++에서 일반적으로 선언되는 방법과 비슷하다. 그러나 배렬의 크기는 콤파일할 때 고정되지 않는다. 머리부파일 <ctype.h>에는 크기지정에 사용되는 size_t 형이 정의되여 있다.

클라스 Vector의 구체례는 그림 20-1에서 보는것처럼 10개의 벡토르요소를 가진다.

the_vec_size [10]

the_p_item

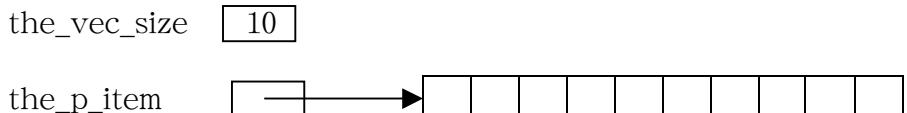



그림 20-1. 클라스 Vector 구체례의 기억기

클라스는 구축자와 해체자로 기억기를 정확히 할당, 해제한다. 클라스 Vector의 구체례는 값주기될수 없다.

20.2.3 클라스 Vector의 실현부

정적성원함수 set_def_size는 기정의 벡토르크기를 설정한다. 벡토르크기에 대한 이러한 기정설정은 실례로 Vector 형객체의 벡토르를 만들 때 리용된다. 개수가 고정되지 않은 객체들이 창조될 때 객체들의 구축자에 대한 파라메터를 명시적으로 설정하는것은 불가능하다.

```
#include <string>
#include <strstream>
#include <stdexcept>

template <class Type>
size_t Vector  <Type>::the_def_size = 0;

template <class  Type>
void Vector <Type>::set_def_size(size_t s)
{
    the_def_size=s;
}
```

구축자는 벡토르의 기억기를 할당하는 보호부(protected)성원함수 new_vec 를 사용한다.

```
template  <class  Type>
Vector <Type>::Vector ( size_t s )
{
   if  ( s > 0 )                                      // 크기
        the_vec_size = s;
  else
          the_vec_size = the_def_size;
  the_vec_actual_size = the_vec_size;            // 공간할당
  the_p_item = NULL;                             // 실패
  the_p_item = new_vec( the_vec_size );
}
```

주의: 구축자의 기정값이 사용되면 그때 실제적인 요소들의 할당은 the_def_size 로 정의된다.

해체자는 요구된 기억기를 해방한다.

```
template <class Type>
Vector  <Type>::~Vector()
{
   if ( the_p_item  ! = NULL )                    // 해방
      delete[ ]  the_p_item;
}
```

개별적요소에 대한 접근은 다중정의연산자 []를 사용한다.

```
template  < class  Type>
Type&  Vector  <Type>::operator[ ] ( size_t  i)
{
```

```
    if ( i < 0 || I >=  the_vec_size) {              // 첨수검사
        fail ( “subscript  is”,  i);                 // 실패
        return   the_p_item[0];                      // 콤파일러검사
    }
    return  the_p_item[i];                           // 성공
}
```

흥미 있는것은 객체 그자체가 아니라 선택된 객체의 주소가 돌려 지는것이다.

```
Type&  Vector  <Type>:: operator[ ] ( size_t  i )
```

따라서 식의 LHS에 첨수를 사용할수 있다. 값주기연산자의 LHS에 있는 항목에 대해서는 왼쪽값(lvalue)이 돌려 지게 된다. 그러나 콤파일러는 연산자가 값주기연산자의 RHS에 사용된다면 이것을 오른쪽값(rvalue)으로 변환할것이다.

주의: 보통 다중정의연산자 []이 inline으로 선언되여 식의 RHS에 사용되면 콤파일러는 돌림결과에 대한 변환 rvalue->lvalue->rvalue를 최량화한다.

클라스의 고정구체례로 사용되는 메쏘드의 const판은 다음과 같다.

```
template  <class Type>
const Type& Vector  <Type>::operator[] (size_t  i) const
{
   if  ( i <0 ||  i > = the_vec_size ) {              // 첨수검사
      fail( “subscript  is”, i ) ;                    // 실패
      return   the_p_item[0] ;                        // 콤파일러검사
    }
    return  the_p_item[i];                            // 성공
}
```

주의: 이것은 고정객체의 참조가 돌려 지기때문에 요구된다.

사용자는 자기가 사용할수 있는 구조체에로의 효과적인 접근을 요구한다.

```
template   <class  Type>
Type&  Vector  <type>:: operator() (size_t  i)
{
   return  the_p_item[i];                             // 빠르다
}
```

inline으로 선언하면 일반적으로 내장된 C++배렬을 사용할 때처럼 같은 코드가 생긴다. 메쏘드의 const판은 다음과 같이 실현된다.

```
template <class Type>
const Type&  Vector<Type>::operator()( size_t i ) const
{
   return  the_p_item[i];                                    // 빠르다
}
```

성원함수 size 는 vector 에서 능동세포들의 수를 돌려 준다. 이것은 기억기의 할당과 다를수 있다.

```
template  <class  Type>
size_t Vector  <Type>:: size( ) const
{
     return  the_vec_size;                // 기억된 요소들
}
```

성원함수 push_back 는 끝에 추가요소를 붙이는것으로 벡토르의 크기를 변경한다. 이것은 새로운 배렬을 창조한 다음 낡은 내용들에 새롭게 할당된 기억기의 추가적인 요소들을 추가복사하여 실현된다. 보다 효률적으로 처리하기 위해 함수 push_back 는 INC_SIZE 요소를 추가하며 추가된 요소를 기억기에 간단히 복사한다.

```
template  <class  Type>
void  Vector  <Type>::push_back (const  Type&  val)
{
  if  ( the_vec_size +1 > the_vec_actual_size )              // 필요한 기억기가 없다
  {
     Type*  old_vec = the_p_item;                             // 초기화

     the_vec_actual_size = the_vec_size + INC_SIZE;     // 새 크기
      the_p_item = new_vec( the_vec_actual_size );
      for ( size_t  i = 0; i < the_vec_size; i ++)
      {
            the_p_item[i] = old_vec[i];                       // 배치하다
      }
       the_p_item[the_vec_size] = val;                        // 기억
       the_vec_size ++;                                       // 크기조절
}
```

성원함수 pop_back 는 벡토르의 끝요소를 삭제하는것으로 벡토르의 크기를 변경한다. 이때 배렬에 의해 형성된 실제적인 기억기를 줄이지는 않는다.

```
template  <class  Type>
void  Vector  <Type>::pop_back()
{
```

```
  if  ( the_vec_size > 0 )
  {
        the_vec_size --;
   }  else  {                              // 크기조절
        fail( "pop_back" ,0);
    }
}
```

벡토르의 첫번째 요소와 마지막요소는 각각 성원함수 front 와 back 를 사용하여 직접 호출될수 있다. 집합으로부터 일부 불필요한 항목의 복사를 피하기 위해 선택된 항목의 참조가 돌려 진다. 매 메쏘드에는 클라스 Vector 의 고정구체례와 가변구체례가 처리될수 있도록 2 가지 판이 존재한다.

```
template  <class  Type>
const  Type&  Vector  <Type>::front ( ) const
{
  return  operator [ ]  (0);                         // 복귀
}

template  <class  Type>
Type&  Vector  <Type>::front( )
{
  return  operator  [ ]  (0);                        // 복귀
}

template  <class  Type>
const  Type&  Vector <Type>::back( )  const
{
  return  operator  [ ]  (the_vec_size-1) ;        // 복귀
}

template  <class  Type>
Type&  Vector  <Type>::back( )
{
  return  operator  [ ]  (the_vec_size-1) ;        // 복귀
}
```

주의: 성원연산자함수 [ ]의 명시적인 호출을 사용한다. 연산자함수 []의 호출

return operator [ ] (0) ;

은

return (*this) [0] ; 으로도 쓸수 있다.

성원함수 new_vec 는 벡토르의 내용을 보유하는 기억기를 할당한다.

```
template  <class  Type>
Type*  Vector<Type>::new_vec ( size_t  n )
{
```

```
  Type*  p_store;
  p_store_new   Type[n] ;                    // 할당

 // 경고
 // 이것은 기억기부족으로 실패할수 있다
 // 만일 그렇다면 레외 bad_alloc 가 발생된다

   return  p_store;                          // 기억기
}
```

성원함수 fail 은 레외를 발생시킨다.

```
template <class Type>
void Vector <Type>:: fail( const char mes[ ], size_t  i ) const
{
 char  storage[120];                                          // 통보문
  std::ostrstream  text (storage, 120);                       // 문자흐름으로
  text << "Vector  [Bounds  0.."<< ( (int) the_vec_size-1 )
        << "- "<< mes << " "<< i << " ] "  << "\0";
   throw  std:: range_error ( std :: string (storage) );        // 레외
}
```

20.2.4 종합서술

벡토르를 만들고 부당한 접근을 시도하는 소규모의 시험프로그람은 다음과 같다.

```
#include <iostream>
#include <string>
#include <stdexcept>
#include " Vector.h"

int  main ()
{
  try
  {
       const MAX = 9;
        Vector  <int>  v(MAX);
        int  i;

       for ( i = 0;  i < MAX ;  i++ )  { v [i] = i;  }                   // 만들기
       for ( i = 0;  i < MAX ;  i++ )  std::cout  << v[i] << "  ";    // 검사
       std::cout << "\n";
       for ( i= 0; i < MAX ; i++ )   std::cout << v(i) << "  ";      // 검사를 안한다
       std::cout << " \n";

       std::cout << " \n" << " \n";
```

```
        std::cout << " Now access element 100 which does not exist" << " \n" ;
        std::cout << v[100];
    }
    catch (std:: range_error& err)
    {
        std:: cout << "\n" << " Fail:" << err. what( ) << " \ n" ;
    }
    return 0 ;
}
```

실행결과는 다음과 같다.

```
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
Now access element 100 which does not exist
Fail: Vector [Bounds 0..8 - subscript is 100]
```

20.2.5 벡토르구체례를 참조로 전달

벡토르의 구체례는 복사(20.2.6 을 보시오.)될수 없어도 다음의 실례처럼 참조에 의하여 함수에 전달될수 있다.

```
int sum (Vector <int> & numbers )
{
  int sum = 0;
  for ( int i = 0 ; i < numbers.size( ) ; i++) sum + = numbers [i] ;
  return sum;
}
int main ()
{
   Vector <int> room(4) ;
    Room [0] = 3 ;                                  // 사람 수
    Room [1] = 2 ;                                  // 사람수
    Room [2] = 1 ;                                  // 사람수
    Room [3] = 1 ;                                  // 사람수
    std:: cout << "Number of people in the rooms is :"
               << sum (room) << "\n" ;
    return 0;
}
```

실행결과는 다음과 같다.

```
Number of people in the rooms is :7
```

20.2.6 실현부의 제한

클라스 Vector 의 실현부는 다음의 제한성을 가진다.

- 용기클라스 Vector 구체레에 값주기를 하면 그 결과들은 정확치 않다. 23.1 과 23.4 에 풀이와 함께 작용결과를 상세히 서술한다.
- 구축자나 해체자를 가지는 클라스항목이 기억되는 경우 그 배렬이 확장될 때 일부 불필요한 구축자와 해체자들이 실현된다.

두번째 제한성에 대한 풀이는 기억기를 본래의 기억기로 할당하는것이다. 단지 집합에 추가될 때에만 기억된 객체의 구축자를 명시적으로 호출할수 있다. 처리에서의 이러한 개선은 메쏘드 ~Vector, push_back, pop_back, new_vec 의 변경을 요구한다.

기억기가 본래의 기억기로 할당되므로 기억된 항목의 해체자는 반드시 클라스 vector 의 해체자에서 명시적으로 호출되여야 한다. 이것은 다음과 같이 실현된다.

```
template  <class Type>
Vector  <Type>:: ~ Vector ()
{
   for ( int  i = 0;  i < the_vec_size;  i ++ )
    {
       (&the_p_item[i] ) -> ~Type ( )  ;          //  해체자호출
     }
     if  ( the_p_item != NULL )                   // 할당 안됨
         delete [ ]  ( char* )  the_p_item ;        // 바이트로 취급
}
```

메쏘드 push_back 의 실현부에는 다음의 연산자 new 가 쓰인다.

```
void*  operator new ( size_t, void* p )           // 초기화
{
     return  p;
}
```

이것은 우에서 본 new 의 다중정의판을 거쳐 기억되여야 할 항목의 구축자를 명시적으로 호출하는것이다.

```
template  < class  Type>
void  Vector  <Type> :: push_back(const  Type&  val)
{
  if  (the_vec_size+1>  the_vec_actual_size)                 // 필요한 기억기가 없다
  {
     Type*  old_vec = the_p_item;                              // 초기화
```

```
    the_p_item  = new_vec (the_vec_actual_size+INC_SIZE);
    for ( int  i = 0; i < the_vec_size;  i++)
         new  ( &the_p_item [i] )  Type (old_vec[i]);           // 구축자

    the_vec_actual_size += INC_SIZE;                            // 새  크기
    for  (int  i = 0;  i < the_vec_size; i ++)                  // 해체자
             (&old_vec[i]) ->~Type ( );                         // 매 요소에 대하여
    if  (old_vec  != NULL )
          delete [ ] (char*) old_vec;                           // 본래의 기억기
      }
      new (&the_p_item[the_vec_size] )  Type (val);             // 구축자
     the_vec_size ++;                                           // 크기조절
}
```

주의: 집합의 초기성원들은 다중정의연산자 new를 사용하여 새롭게 할당된 기억기에 복사된다. 이 기구는 해당한 구축자처리를 수행하기 위해 요구된다.

메소드 pop_back는 삭제항목들의 해체자를 명시적으로 호출한다.

```
template  <class Type>
void  Vector <Type>::  pop_back( )
{
  if ( the_vec_size > 0 )
  {
     ( &the_p_item[the_vec_size-1] ) -> ~Type( );    // 해체자호출
      the_vec_size -- ;                              // 크기조절
   } else { fail(“pop_back”, 0”; )                   // 무효
}
```

기억기의 본래바이트들을 위한 기억기할당은 다음과 같다.

```
template  <class Type >
Type*  Vector<Type>:: new_vec  ( size_t  n )
{
   Type*  p_store;

    // 경고
    // 이것은 기억기부족으로 실패할수 있다
    // 례외가 발생하면 bad_alloc가 처리될것이다

    p_store  = ( Type* ) new char[n * sizeof (Type)];      // 할당
    return   p_store;                                      // 기억기
}
```

20.3 Vector 클라스에 의한 탄창의 실현

8.9 에서 서술된 클라스 Stack 는 자료항목들의 기억기를 제공하는 용기클라스들을 사용하여 다시 실현될수 있다. 메쏘드 size, push_back, pop_back, back 를 실현하는 용기클라스인 경우에는 그것이 리용될수 있다. 따라서 용기클라스 vector 는 이 메쏘드들을 실현하므로 리용될수 있다. 자료기억기를 위한 용기클라스를 사용하는 클라스 Stack 의 명세부는 다음과 같다.

```
#ifndef  CLASS_STACK_SPEC
#define  CLASS_STACK_SPEC

template  <class  Type, class  container>
class Stack{
public:
   typedef   Container:: Value_type  Value_type;   // 용기에서
    stack();
    inline bool  empty ( )   const;                       // 빈 탄창
    inline size_t  size ( ) const                         // 집합의 크기
    inline Value_type&  top( );                           // 꼭대기항목을 돌려 준다
    inline const  Value_type&  top( )   const ;           // 꼭대기항목을 돌려 준다
    inline void  push ( const Value_type) ;               // 탄창우에 항목을 넣는다
    inline void   pop( );                                 // 탄창에서 꼭대기항목을 뽑는다
private:
   Container  the_elements;                               // 기억기
};
```

주의: 용기안에 value_type 를 만들기 위해 형정의 typedef 를 한다. 용기안에 기억된 객체들의 형은 value_type 일것이다.

클라스 Stack 의 실현부는 다음과 같다.

```
template  <class Type,  class  Container>
Stack<Type,  Container>::  Stack()
{
}

template  < class Type, class Container>
bool Stack<Type,  Container>::empty ( )   const
{
  return  the_elements.size ( )  <= 0;                 // 비다
}

template  <class  Type, class Container >
size_t Stack<Type,  Container>::size ( ) const
{
  return  the_elements.size();                         // 탄창안의 요소들
```

```
}
template <class  Type, class  Container>
Value_type&  stack<Type,Container>::top ( )
{
  return the_elements.back ( );                          // 꼭대기항목
}
template <class  Type, class  Container>
const  Value_type&  Stack<Type,  Container>::top() const
{
    return  the_elements. Back();                        // 꼭대기항목
}

template <class  Type,  class Container>
void   Stack<Type,  Container>::push (const Value_type  item)
{
  the_elements.push_back(item);                          // 탄창에 넣기
}
template <class  Type, class Container>
void Stack<Type,  Container>::pop( )
{
 the_elements.pop_back( );                               // 삭제
}
#endif
```

20.3.1 종합서술

용기클라스 Vector 를 사용하여 int 형탄창을 다음과 같이 선언한다.

```
Stack<int, Vector<int> > numbers;
```

주의: Stack <Vector<int> >안에서 >와 >사이에 공백을 주지 않으면 콤파일러는 연산자 >>를 사용하는 구조로 혼돈한다.

용기클라스 Vector 를 사용하여 char 형탄창을 다음과 같이 선언한다.

```
Stack < char,  Vector <char> > letters;
```

기억기를 위한 다른 용기클라스는 메쏘드 size_push_back, pop_back, back 를 실현하는 경우에만 리용될수 있다. 실례로 실현부가 련결목록을 리용하고 우에서 말한 대면부를 제공하는 용기클라스 List 를 들수 있다. 이때 우의 기능을 리용한 련결목록을 사용하여 옹근수의 탄창을 다음과 같이 선언한다.

```
Stack<int,  List<int> >   number;
```

20.3.2 본보기클라스의 기정파라메터

본보기클라스의 인수에도 함수에서와 같이 기정파라메터를 줄수 있다. 실례로 클라스 Stack는 다음과 같이 정의될수 있다.

```
template<class  Type, class Container = Vector<Type>  >
class  Stack  {
public:
}
```

요소들을 가지고 있는 클라스 Vector구체례의 선언은 다음과 같다.

```
Stack <char>  letters;
```

20.4 하쉬표

재정의될수 있는 연산자 []는 전통적인 배렬들과 관련된 새로운 구조체들을 리용한다. 일부 응용프로그람들에서는 첨수(index)나 임의의 문자렬형태를 가질수 있는 열쇠(key)들을 가지는 배렬이 요구된다.

실례로 서로 다른 색갈의 많은 승용차들이 매일 얼마나 생산되는가를 기록하는 차제조와 관련된 응용프로그람에서 하쉬표(Hashtable)는 유용하게 쓰일수 있다.

```
std::cout << “Number of Green cars = ” << cars[“green”] << “ \n”;
```

이것은 열쇠를 첨수로 변환하는 하쉬법을 리용하여 실현될수 있다. 이때 변환된(hashed)열쇠는 세포들의 사슬에 대한 지적자들의 하쉬표에 접근하는데 리용되며 이 지적자들은 쌍(“green”,20)을 가진다. 이 경우 첫번째 요소는 열쇠이고 두번째 요소는 열쇠와 관련된 값이다.

하쉬한다는것(hashing)은 간단히 말하면 열쇠를 구간(0에서부터 하쉬표의 최대요소수에서 1을 던 값에 이르는)의 어떤 수에로 변환한다는것이다. 이때 하쉬된 열쇠는 하쉬표에 접근하는데 리용된다.

하쉬값은 하나가 아니므로 열쇠와 자료값은 같은 값에로 하쉬하는 항목들의 사슬형태인 하나의 쌍으로 기억된다. 이렇게 하면 하나의 요소에 여러개의 열쇠들이 하쉬된 경우 그 요소에 접근될 때 정확한 열쇠가 선택되게 할수 있다.

실례로 다음의 코드를 보기로 하자.

```
cars[ “green ” ]  = 2;   cars[ “blue ” ] = 3;
```

그림 20-2에서는 2개의 열쇠문자렬 (green, blue)을 하쉬값 4로 변환한 경우 매 색갈의 차개수들을 기억하는데 사용된 구조를 설명한다.

green 2 → blue 3

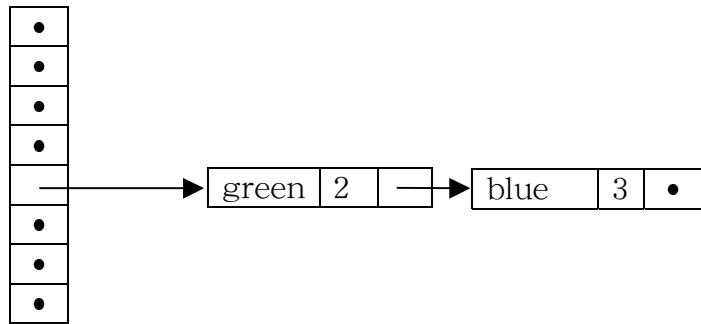



그림 20-2. 같은 하쉬값을 가지는 2개 항목을 위한 하쉬표.

클라스 Hashtable의 명세부는 다음과 같다.

메쏘드/구축자	책 임
Hashtable	객체에 대한 기억기를 할당한다
[ ]	열쇠와 련관된 자료를 호출한다

사용자가 열쇠와 자료의 형을 지정할수 있는 클라스 Hashtable의 명세부는 다음과 같다.

```
# ifndef   CLASS_STRING_ VEC_SPEC
# define   CLASS_STRING_VEC_SPEC
template <class Key,  class data >
class Hashtable  {
public :
     explicit  Hashtable (const unsigned int, unsigned long  (*f ) (const key& )  );
      ~Hashtable ( ) ;                                           // 기억기해방
      Data& operator [ ] (const  key )  const ;                  // 첨자
      const  Data&  operator [ ] (const key )  const ;           // 첨자상수
protected :
      void fail (const char mes [ ] ,  const int ) const ;
private :
       struct  Cell ;                                             // 림시선언
        typedef  Cell  *p_Cell ;
        struct Cell  {
            key     key_value ;                                   // 열쇠값
            Data     value ;                                      // 자료값
            p_Cell   p_next ;                                     // 목록에서 다음항목
        }
       p_Cell  *the_p_cell_vec ;                                 // p_Cell에 대한 지적자
        int        the_no_elements ;                              // 요소들의 번호
        unsigned long (* the_hash_fun ) (const key&)              // 하쉬함수지적자
};
#endif
```

그림 20-3 에서 클라스 Hashtable 로 조작된 구조를 설명한다.

the_p_cell_vec

Cell

green | 2

blue | 3

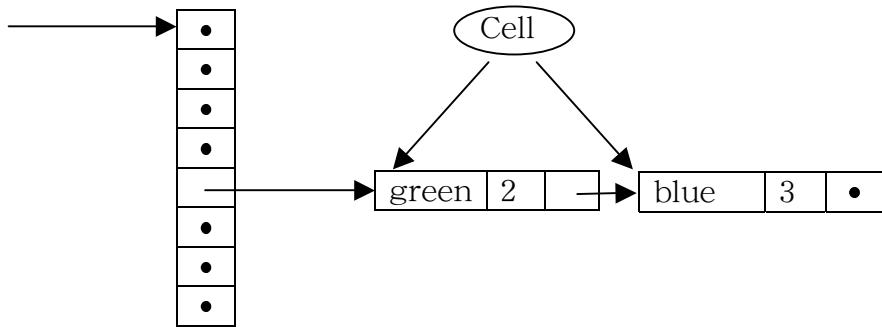



그림 20-3. 클라스 Hashtable 에 의해 사용된 기억기

주의: 세포의 선언은 항목들이 개별적으로 기억된 기억기를 서술한다.

클라스 Hashtable 의 구축자는 다음의 파라메터들을 가지고 있다.

- 사용하려는 하쉬표의 크기
- unsigned long f (const key&)로 서술되는 함수 f 의 주소 (이 함수는 key 를 옹근수로 변환한다.)

클라스 Hashtable 의 실현부는 다음과 같다.

```
#ifndef  CLASS_STRING_VEC_IMP
#define  CLASS_STRING_VEC_IMP

#include <string>
#include <strstream>
#include <stdexcept>
template <class Key, class Data>
Hashtable<Key, Data>::Hashtable (const unsigned int size,
                                 unsigned long (*f) (const Key&) )
{
  the_no_elements = size;

  if ( the_no_no_elements<1)
      fail( "Invalid bounds –", the_no_elements);    // 하쉬크기는 무효

  // 경고
  // 이것은 기억기부족으로 실패할수 있다
  // 만일 그렇다면 례외 bad_alloc 가 발생된다

  the_p_cell_vec = new p_Cell [the_no_elements];  // 하쉬표

  for(int i = 0;  i < the_no_elements; i++)     // 하쉬표안의 지적자들을 null 로 설정
    the_p_cell_vec[i] = NULL;
```

```
    the_hash_fun = f;                                    // 하쉬함수의 주소
}
```

주의: 하쉬표크기는 사용된 단일한 열쇠들의 개수의 크기와 비슷한 씨수이다.

해체자는 창조된 모든 기억기를 해방한다.

```
template <class Key, class Data>
Hashtable <Key,Data>::~Hashtabe( )
{
    for(int i = 0; i < the_no_elements;  i++)            // 모든 목록들
   {
       p_Cell p_cell = the_p_cell_vec[i];
       while  (p_cell  != NULL)                          // 목록
      {
          p_Cell p_current = p_cell;
          p_Cell = p_Cell->p_next;                       // 사슬안의 다음세포
          delete [ ] p_current;                          // 현재세포
      }
   }
    delete [ ] the_p_cell_vec;                           // 하쉬표를 해방
}
```

fail 함수는 다음과 같이 정의된다.

```
template <class Key, class Key, class Data>
void Hashtable<Key,Data>::fail (const char mes[ ], const int i ) const
{
     char storage[120];
     std::ostrstream text(storage,120);
     text<< “Hashtable [ ” << mes << : : << i  << “ ] ” <<  “ \0 ”;
     throw std::range_error (storage );
}
```

주의: 이것은 기억기가 할당될수 없을 때에만 호출된다.

다중정의연산자 []는 련관된 배렬을 첨수화하는데 사용된다. 아래에 보여 준 설명은 변하기 쉬운 객체들을 위한것이다. 하쉬표안에 없는 열쇠가 사용되면 그 열쇠에 대한 새로운 내용이 창조된다.

```
template <class Key, class Data>
Data& Hashtable<Key, Data>::operator[] ( const Key Key_value )
{
     unsigned int h =(*the_hash_fun) (key_value)%the_no_elements;
     P_Cell *p = &the_p_cell_vec[h];                 // p-Cell 에 대한 지적자
```

```
    while (*p !=NULL) {
        if ( (*p)->key_value == key_value ) {
            return (*p)-> value;                        // 세포를 찾는다
        }
        p = &( (*p)->p_next;                           // 목록에서 다음
    }

  // 새로운 열쇠가 세포를 만들고 사슬안에 련결된다

  // 경고
  // 이것은 기억기부족으로 실패할수 있다
  // 그 경우에는 례외 bad_alloc 가 내보내 진다

    *p = new Cell[1];                                  // 세포구성
    (*p)->key_value = key_value;                       // 열쇠를 기억
   (*p)->p)_next      = NULL;
    return (*p)->value;                                // 자료값(정의안됨)
}
```

주의: 열쇠요소가 접근되고 미리 기억되여 있지 않다면 그에 대한 내용은 정의되지 않은 값으로서 창조된다. 이것은 이 정의되지 않은 값이 사용된다면 후에 오유를 발생시킬수 있다. 이것은 어떤 형의 하쉬표가 창조될수 있도록 한다. 보다 좋은 기구는 동일한 요소와 값이 없는 형을 사용하는데 이를 통하여 모든 형들이 같거나 값이 없다는것을 알수 있다.

이 메쏘드의 const 판은 다음과 같다.

```
template <class Key, class Data>
const Data& Hashtable<Key,Data>::operator[ ] (const Key key_value)
const
{
  unsigned int h = (*the_hash_fun) (key_value)%the_no_elements;
  p_Cell *p = &the_p_cell_vec[h];            // p_Cell 에 대한 지적자

  while (*p !=NULL ) {
    if ( (*p)->key_value == key-value ) {
      return (*p)->value                      // 세포를 찾는다
    }
    p = & ( (*p)->_next);                     // 목록에서 다음
  }
  //고정객체안에서 요소를 창조할수 없다

 fail ( "Element undefined (Reading const hashtable) ",0);
 return (*p)->value;                          // 여기서는 얻어 지지 않는다
}

#endif
```

주의: 고정객체의 비존재성원에 대한 접근이 시도되면 실패가 발생한다.

20.4.1 종합서술

사용자는 다음과 같이 쓸수 있다.

```
int main ( )
{
  Hashtable<std::string, int> cars (11, hash);

  cars[ "green" ] = 20;  cars[ "blue" ] = 30;

  cars[ "green" ]++;
  cars[ "blue" ] = cars[ "blue" ]+3;

  std::cout<< "Number of Green cars = " << cars[ "green" ]<< "\n" ;
  std::cout<< "Number of Blue cars = " << cars[ "blue" ]<< "\n" ;
  return 0;
}
```

여기서 하쉬함수 hash 는 다음과 같다.

```
unsigned long hash ( const std::string& str )   // 번호에 대한 하쉬문자렬
{
   unsigned long h = 0;
   for ( int i = 0; i < str.length( ); i++ )
   {
      h = h + (unsigned long) str[i];              // 문자렬에서 총 문자렬
    }
    return h;
}
```

주의: 하쉬표범위내에 있는 하쉬값에 대한 조절은 첨수화를 진행하는 동안 수행된다.

해당머리부파일과 함께 콤파일될 때 실행결과는 다음과 같다.

```
Number of Green cars = 21
Number of Blue   cars = 33
```

20.4.2 성긴 배렬

클라스 Hashtable 은 또한 성긴 배렬(sparse array)들을 실현하는데 사용된다. 실례로 성긴 옹근수배렬을 사용하는 프로그람은 다음과 같다.

```
unsigned long hash_index( const unsigned long& value )
{
```

```
  return value;
}
int main ( )
{
 Hashtable<unsigned long, int> sparse_array(17,  hash_index );
 sparse_array[ 12345 ]       =56;
 sparse_array[123456789] =42;
 std::cout << sparse_array[ 123456789 ] << "\n" ;
 return 0;
}
```

20.5 자체평가

- 클라스 Vector 의 구축자는 왜 명시적으로 선언되는가?
- 클라스 Vector 는 왜 선택된 객체의 참조를 돌려 주는가?
- 프로그람작성자는 왜 클라스 Vector 의 구체레복사를 성과적으로 할수 없는가?
- 만일 클라스 Hashtable의 구체레가 1개 요소를 가진 하쉬표를 가진다면 여전히 많은 자료항목들을 기억할수 있는가?

20.6 련습

- 크기변경가능한 배렬
  사용자가 배렬의 크기를 변경할수 있도록 클라스 Array 를 변경하시오. 배렬의 크기를 변경할 때 초기배렬과 크기변경된 배렬에 공통적인 세포들의 초기내용은 보존된다. 이 보존은 복사에 의해 실현될수 있다.
  배렬의 구체레는 복사되지 않는다는것을 기억하시오.
- 바른 6 면체
  3 차원을 가지는 안전한 배렬을 제공하는 클라스를 작성하시오.
- 2 진나무
  2 진나무를 리용하여 열쇠, 자료쌍을 기억하고 검색하는 클라스를 작성하시오.

21 전처리지령

이 장에서는 C++전처리기(preprocessor)로서 제공되는 마크로언어에 대하여 서술한다. 마크로는 본질에 있어서 프로그람기호들의 렬을 다른 기호들의 묶음으로 재배치하는것이다. 마크로는 정확히 사용될 때 매우 강력한 기능을 나타낸다. 그러나 정확히 사용되지 못하면 프로그람을 해석할수도 없고 수정할수도 없게 된다.

21.1 소 개

마크로는 하나의 프로그람토큰(token)들의 묶음을 다른 토큰들의 묶음으로 바꾸는 본문치환(textual substitution)이다. 이러한 마크로처리는 원천본문이 해당콤파일러에 의해 처리되기전에 수행된다. 마크로는 본문파일의 1렬에 있는 #에 의하여 정의된다. C++에서는 많은 경우 마크로를 쓰지 않고 내부전개(inline)기능과 상수선언을 리용한다.

21.2 파일원천포함

마크로처리기의 기능의 하나는 파일내용들의 원천을 프로그람본문안에 포함시키는것이다.

```
#include “ filename”          /*현재작업등록부에서 취해 진다*/
#include <filename>           /*체계등록부들에서 취해 진다 */
```

21.3 본문치환

본문치환은 대부분의 경우 상수선언들과 내부전개기능을 리용하여 실현된다. 다음의 명령문은 기호 MAX를 120으로 치환한다.

```
#define    MAX 120
```

주의: 마크로의 유효범위는 마크로가 정의된 곳으로부터 콤파일단위의 끝까지이다. 반두점(;)은 문제를 일으킬수 있으므로 마크로정의의 마지막에 놓지 않는다.

실례로 코드

```
#define MAX  120;      // #은 반드시 1렬에 있어야 한다
int  table[MAX];
```

는 콤파일하면

```
int table [120;];
```

로 될수 있는데 점찍기가 명백치 않으므로 콤파일시 오유를 발생시킨다.
마크로정의는 다음의 선언으로 바꿀수 있다.

```
#undef  MAX
```

앞으로는 기호 MAX를 120으로 바꾸는 작업을 하지 않겠다.
다음의것은 마크로정의인데 x와 y가운데서 큰것을 넘겨 주는 조건식을 사용한다.

```
#define  larger( x, y )  ( (x) > (y) ? (x) : (y) )
```

주의: 마크로를 정의할 때 괄호들의 사용에 주의하시오.

실례로 larger가

```
#define larger (x,y)  x > y  ?  x : y
```

와 같이 정의되면 본문

```
Max=larger(count+1,last) * 2;
```

가 다음과 같이 치환된다.

```
Max = count + 1 > last ? count + 1 : last * 2;
```

주의: 결과가 거짓일 때 더 큰것으로 돌려 지는 결과만이 2로 곱해 진다.

보다 유용한것은

```
#include <stdlib>
#define assert (ex)
{
     if ( ! (ex) )
     {
         std :: cerr <<"assert failed File  "  <<_FILE_<<  " line  " ;
         std ::cerr <<_LINE_<<  "\n  " ;
         std ::exit ( -1 );
      }
}
```

로서 정의될수 있는 assert마크로인데 이것은 진행중의 프로그람안에 놓여 프로그람을 실행하는 동안 여러가지 조건들을 주장한다. 실례로 5.3.1에서 본 클라스 Account의 메쏘드 withdraw는 아래와 같이 실현될수 있다.

```
float Account ::withdraw (const float money)
{
   assert (money >= 0.00);                       // 사전조건
   float  get = 0.00;
   if ( the_balance-money >= the_min_balance )
   {
       the_balance = the_balance-money;
       get = money;
    }
    assert ( get >= 0.00 );                      // 사후조건
    return  get;
}
```

프로그람이 정확히 동작하면 마크로는 다음과 같이 변화된다.

```
#define assert (ex)
```

이렇게 하면 프로그람이 재콤파일될 때 프로그람크기를 상당히 줄일수 있다. 물론 이것은 프로그람에 대한 검사시간도 빨라 진다는것을 의미한다.

주의: 마크로가 두행이상으로 전개되면 \을 써서 마크로의 내용이 계속된다는것을 가리킨다.
_FILE_은 현재파일이름으로 교체되는 체계마크로이다.
_LINE_은 현재행번호로 교체되는 체계마크로이다.
마크로 assert는 머리부파일 <assert>안에 정의된다.

21.4 조건부콤파일

조건부콤파일(conditional compilation)을 사용할 때는 심중해야 한다. 조건부콤파일은 응용프로그람작성자에게 그 프로그람의 각이한 판본들이 같은 원천파일에 포함될수 있게 하는 강력한 기능을 제공하지만 코드가 실제적으로 어떤 특수한 판본을 콤파일하게 하는 혼란을 가져 올수 있다.

실례로

```
#include <iostream>
#include  "local.h"

int main()
{
#ifndef  DEBUG
     std ::cerr <<  "Entering function main"  << " \n";
```

```
#endif
      std::cout << “Hello world” << “ \n” ;
      return 0;
}
```

에서 파일 local.h가 포함되고

```
#define DEBUG   1
```

이 들어 있으면 C++콤파일러는 본문을 다음과 같이 콤파일한다.

```
#include <iostream>
#include  “local.h”

int main()
{
     std :: cerr << “Entering function main”  <<  “ \n” ;

     std :: cout << “Hello world”  <<  “ \n” ;
     return  0;
}
```

반대로 파일 <local.h>에 기호 DEBUG의 정의

```
// #define DEBUG   1
```

가 들어 있지 않으면 C++콤파일러는 본문을 다음과 같이 콤파일한다.

```
#include <iostream>
#include  “local.h”

int main()
{
        std::cout <<  “Hello world”  <<  “ \n” ;
         return   0;
}
```

조건부콤파일에서 중요한것은 본문이 해당한 C++콤파일러에 의해 콤파일되기전에 치환이 진행되는것이다.

다른 조건들을 실례로 들수 있다.

```
//define   MAX    10

#ifndef   MAX
    std ::cout <<  “MAX not defined” ;
```

```
#endif
```

이것은 통보문

```
MAX not defined
```

을 인쇄하기 위한 코드를 콤파일한다.
조건은

```
#define Intel  7
#ifdef  Mips
    std :: cout << “Mips CPU” << “\n”;
#elif  Intel
    std :: cout << “Intel CPU” << “\n”;
#else
    std :: cout << “Not a Mips or Intel CPU” << “\n”;
#endif
```

에서와 같이 else부분을 가질수 있다.
이것은

```
Intel CPU
```

을 인쇄하기 위한 코드를 콤파일한다.
조건은 또한

```
#define Intel  8
#if  Intel = = 7
   std :: cout << “p7 CPU” << “\n”;
#elif Intel == 8
   std :: cout<< “p8 CPU” << “\n”;
#else
   std :: cout << “Intel CPU” << “\n”;
#endif
```

에서와 같이 정의된 기호들로 이루어 진 콤파일시 조건일수도 있다.
이것은

```
P8  CPU
```

를 인쇄하기 위한 코드를 콤파일한다. 다항연산자 defined를 #if와 함께 사용하여 여러개의 기호들이 정의되였는가를 검사할수 있다.

```
#if defined  Intel && defined Linux
  std :: cout << "Intel CPU running Linux" << "\n";
#endif
```

21.5 error지령

#error지령은 콤파일에서 치명적오유를 일으키는데 사용된다. 실례로 프로그람에서 실행자들은 다음의 코드렬을 통하여 정해 진 기호 MAX가 범위 0–100사이에 있게 할수 있다.

```
#if  MAX < 0 || MAX > 100
#error  MAX must be in range 0 .. 100
#endif
```

콤파일처리부분에서 정해 진 기호 MAX가 규정된 범위밖에 설정되면 콤파일은 다음과 같은 통보문을 내보내고 중지된다.

```
Fatal example.cpp 9:Error directive : MAX must be in range 0..100 in
function main()
```

21.6 pragma지령

실현부의존지령 #pragma는 어떤 콤파일항목들을 설정하는데 사용된다. 실례로 프로그람코드의 공간최량화를 선택하는 pragma지령은 다음과 같다.

```
#pragma space_optimization
```

주의: 이것은 만들어 낸 pragma지령인데 정확한 pragma지령에 대해서는 해당한 콤파일 문서에서 찾아 보면 된다.

콤파일러는 자기가 인식할수 없는 pragma지령들은 무시한다.

21.7 line지령

이 지령은 초기행번호와 파일이름이 전처리를 하기전에 있다는것을 콤파일러에 알려 주는데 사용된다. 이것은 보통 #include지령으로 지정된 원천파일들의 포함을 수행하는 C++전처리기에 의해서 자동적으로 발생된다. 이런 방법으로 콤파일러는 전처리기에 의해 발생된 파일행과 파일이름이 아니라 초기파일들의 원천행번호와 파일이름을 줄수 있다.

```
# line 27
# line 1      "Account.h"
```

21.8 미리 정의된 이름

다음의 표에서는 미리 정의된 값을 가지는 마크로이름들을 소개한다.

이 름	설 명
_cplusplus	콤파일러가 C++콤파일러라면 정의된다
DATE	현재날자(월 일 년)
FILE	현재파일이름
LINE	현재행번호
TIME	현재시간(시:분:초)

주의: _cplusplus는 머리부파일이 C++프로그람과 C프로그람들사이에 공유된것이라면 리용될수 있다.

21.9 문자렬만들기

마크로치환은 하나의 문자렬에 대해서는 진행되지 않으며 문자렬을 동적으로 만들려고 할 때 마크로연산자(#)를 리용한다. #연산자는 마크로내부에서 리용될 때 다음토큰(token)을 문자렬인용부호로 막아 준다. 실례로 그 인수로부터 문자렬을 발생시키는 마크로 AS_STRING은 다음과 같이 정의된다.

```
#define  AS_STRING( name )  #name
std :: cout << AS_STRING( Mike ) << "\n" ;
```

이것은 다음의 출력행을 발생시킨다.

```
std::cout << "Mike" << "\n" ;
```

21.10 토큰들을 함께 붙이기

##연산자는 토큰들을 함께 붙이기 위해 마크로안에서 사용되는데 이때 마크로치환들을 련속적으로 수행할수 있다. 실례로

```
#define  ENGLISH_1     "one"
#define  ENGLISH_2     "two"

#define  FRENCH_1      "un"
#define  FRENCH_2      "deux"

#define  REAL_PASTE ( x, y ) x ## y
```

인 마크로정의들에 대하여 행

```
std :: cout << REAL_PASTE( ENGLISH, _1 ) << "\n" ;
```

을 처리하는 결과는 다음과 같다.

```
std :: cout << "one" << "\n" ;
```

우의 마크로를 다시 교체하여 마크로에 인수를 포함시킬수도 있다. 실례로

```
#define PASTE ( x, y ) REAL_PASTE( x, y )
#define LANGUAGE FRENCH
#define NUMBER_2 PASTE ( LANGUAGE, _2 )
```

를 사용하여

```
std ::cout << NUMBER_2 << "\n" ;
```

을

```
std :: cout << "deux" << "\n" ;
```

로 바꿀수 있다.

21.11 마크로의 다중사용

마크로는 다음의 프로그람안에서와 같이 극단적으로 사용될수 있는데 이 프로그람은 C++를 Pascal이나 Ada형식의 프로그람작성언어와 같은것으로 볼수 있도록 이 언어의 성질을 변화시켰다.

```
#define IF          if (
#define THEN        ) {
#define ELSE        } else {
#define ELIF        } else  if (
```

```
#define FI          ; }
#define BEGIN   {
#define END       }
#define WHILE   while(
#define DO        ) {
#define OD       ; }
```

```
#include <iostream>
int main()
BEGIN
    int  countdown;
    countdown = 4;
     WHILE  countdown >0  DO
           cout << countdown << “\n”;
            IF  countdown = =3 THEN
                 std ::cout << “Ignition” << “\n”;
             FI
             countdown = countdown -1;
       OD
       std ::cout << “Blast off” << “\n”;
       return  0;
END;
```

이 프로그램의 실행결과는 다음과 같다.

```
4
3
Ignition
2
1
Blast off
```

21.12 머리부파일의 포함

C++에서 머리부파일은 보통 다음과 같이 구성된다.

```
#ifndef  CLASS_NAME_SPEC
#define CLASS_NAME_SPEC
//한번만 포함되는 원천으로서의
//클라스명세부
#endif
```

이것은 머리부파일이 프로그람에서 여러번 포함될수 있게 하지만 파일내용에 대한 콤파일러의 처리는 한번만 진행되게 한다.

주의: 이것은 실례로 다음과 같은 상황에서 진행될수 있는데 여기서 프로그람은 하나의 이름을 보유하는 클라스명세부를 포함하는 Car, Person 클라스들의 명세부를 리용한다.

클라스 Person 명세부	클라스 Car 명세부

#include "name.h"
 	
class person {
//클라스명세부
}
	#include "name.h"
	
	class Car {
	//클라스명세부
	}

21.13 파라메터의 개수가 변하는 경우의 명세부

C++에서는 가변개수파라메터들을 가지는 함수를 선언하고 사용할수 있다. 실례로 함수 sum_params 는 자기 파라메터들의 합을 구하도록 씌여 진다. 파라메터목록은 0 값을 가지는 파라메터로 끝난다. 이것은 다음과 같이 실현될수 있다.

```
#include <stdarg.h>
int sum_params( const int , ... );                // 원형
int sum_params( const int first , ... )
{
   va_list  p_arg;                                // 인수에 대한 지적자
   va_start( p_arg , first );                     // 첫번째 지적자를 설정
   int sum = 0,  value = first;
   while ( value != 0)
   {
      sum += value;
      value = va_arg( p_arg , int );              // 목록안의 다음값
   }
   va_end( p_arg );                               // 삭제
   return sum;
}
```

주의: 함수가 가변개수파라메터들을 가진다는것을 지적하기 위해 …을 사용한다.
파라메터목록을 끝내기 위해 0 값을 사용한다.
우에서와 같이 마크로를 사용하자면 적어도 한개의 파라메터가 지정되여야 한다.
이러한 수법을 써서 콤파일러가 제공된 파라메터들의 형과 개수를 검사할수 없다는것을 의미한다.

<stdarg.h>안에 포함된 마크로들은 기계에 의존하지 않고 파라메터에 접근한다. 그의 함수들은 다음과 같다.

va_list	인수목록지적자 p_arg 를 선언한다.
va_start	인수 first 다음의 인수지적자 p_arg 를 초기화한다.
va_arg	옹근수형인 경우에 p_arg 로 지시된 인수를 돌려 준다. 이때 다음인수에로 p_arg 를 전진시킨다.
va_end	후에 삭제한다.

함수 sum_params 는 다음의 프로그람에서 사용된다.

```
int main()
{
    std :: cout << “The sum of 1, 2, 3, 4, 5 is ”;
    std :: cout <<sum_params( 1, 2, 3, 4, 5, 0 ) << “\n”;
    return 0;
}
```

실행결과는 다음과 같다.

```
The sum of 1, 2, 3, 4, 5, is 15
```

주의: 우의 처리에서 오유가 생길 가능성은 전혀 없다.

21.14 자체평가

- C++에서 마크로기능은 대체로 중복해서 쓸수 있는데 왜 그런가?
- 마크로보다 본보기함수를 사용하는것이 왜 더 좋은가?
- 마크로를 언제 사용하는것이 좋은가?

21.15 련 습

- assert_to_file
  주장한 조건이 거짓일 때 파일 file_name 에 문자렬과 어떤 해당한 오유수정 정보를 쓰는 마크로 assert_to_file(condition, string, file_name)을 작성하시오. 파일은 마크로의 요구에 따라 열려 지고 닫겨 진다.

22 C++의 입출력

이 장에서는 C++입출력체계에 대하여 서술한다. 특히 흐름(stream)에 대한 읽기, 쓰기정보의 형식을 변화시키는 조작자(manipulator)들이 어떻게 구성되는가를 보여준다.

22.1 C++입출력체계의 개요

C++입출력체계(I/O system)는 클라스들의 집합에 의해 실현된다. 아래에 입출력체계와 관련한 클라스들의 계승구조(inheritance structure)를 총체적으로 보여 준다.

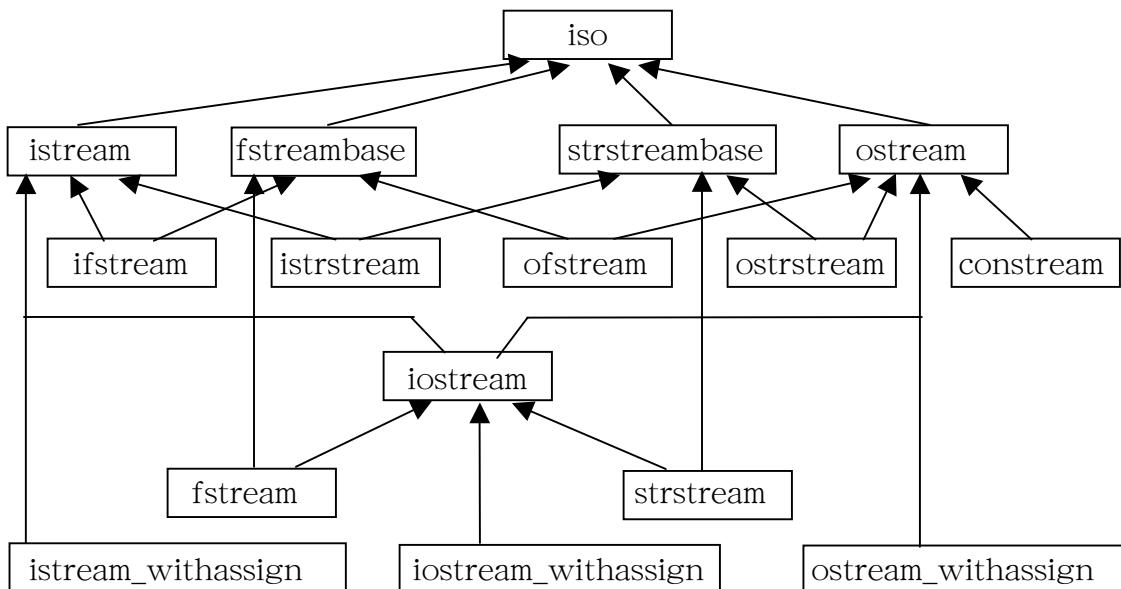



그림 22-1. C++입출력체계의 계승도

22.1.1 클라스 ios

값	설 명	값	설 명
in	파일입력	binary	2진자료
out	파일출력	cur	파일의 현재위치를 탐색
app	파일에 추가	end	파일끝을 탐색
beg	파일시작을 탐색	trunc	창조할 때 파일끝을 자르기

클라스 ios는 입출력클라스들의 모임이며 특히 우의 값들을 정의한다.

22.1.2 클라스 ostream

클라스 ios에서 파생된 클라스 ostream은 표준출력장치에 형식화된 출력을 제공한다. 머리부파일 <iostream>에 객체 cout에 대한 다음의 선언이 있다.

```
extern ostream cout;
```

클라스 ostream은 다음의 메쏘드를 실현한다.

메쏘드	설 명
<<	모든 표준형들에 대한 다중정의추출연산자

22.1.3 클라스 istream

클라스 ios에서 파생된 클라스 iostream은 표준입력장치에 형식화된 입력을 제공한다. 머리부파일 <iostream>에 객체 cin에 대한 다음의 선언이 있다.

```
extern istream cin;
```

클라스 istream은 다음의 메쏘드를 실현한다.

메쏘드	설 명
>>	모든 표준형들을 위한 다중정의삽입연산자

22.1.4 클라스 ifstream과 ofstream

클라스 ifstream과 ofstream은 디스크파일에 대한 입출력을 진행한다. 클라스 ifstream(Input file stream:입력파일흐름)은 클라스 istream에 추가된 다음의 메쏘드들을 실현한다.

메쏘드	설 명
open(name,mode)	파일을 연다. 여기서 파라메터 name은 C++문자렬, mode는 ios::in의 기정값을 가진다.
close()	흐름을 닫는다.
is_open()	흐름을 열면 참을 돌려 준다.

클라스 ofstream(Output file stream:출력파일흐름)은 클라스 ostream에 추가된 다음의 메쏘드들을 실현한다.

메쏘드	설 명
open(name,mode)	파일을 연다. 여기서 파라메터 name은 C++문자렬, mode는 ios::out || ios::trunc이다.
close()	흐름을 닫는다.
is_open()	흐름을 열면 참을 돌려 준다.

22.1.5 클라스 istrstream과 ostrstream

클라스 istrstream과 ostrstream은 프로그람안의 기억기위치들에 대한 입출력을 진행한다. 이 흐름들은 프로그람안에서 내부적으로 사용되는 형식본문에서 쓸모 있다. 클라스 ostrstream은 클라스 ostream의 메쏘드들외에 다음의 추가메쏘드를 가진다.

메쏘드	설 명
str()	입력된 본문의 C++문자렬(char*)을 돌려 준다.

22.1.6 표준흐름

다음의 표준흐름들은 머리부파일 <iostream>안에 정의된다.

char로	wchar_t로	설 명
extern istream cin;	extern wistream win;	표준입력
extern ostream cout;	extern wostream wout;	표준출력
ectern osrteam cerr;	extern wostream werr;	오유출력(완충 없음)
extern ostream clog;	extern wostream wlog;	리력출력

22.2 삽입자와 추출자, 입출력조작자

흐름입출력체계는 입출력이 프로그람안에서 선언된 객체들에 대하여 직접 수행될수 있도록 자체로 정의될수 있다. 실례로 5.3.1에서 정의된 클라스 Account를 사용한 명세부는 다음과 같다.

```
class Account {
public:
    Account ( );
    float account_balance ( ) const;          // 잔고를 돌려 준다
    float withdraw (const float);             // 구좌에서 출금
    void deposit (const float);               // 구좌에 저금
    void set_min_balance(const float);        // 최소잔고를 설정한다
private:
    float the_balance;                        // 현재잔고
```

```
    float the_min_balance;                    // 최소잔고
};
```

다음의 항목들에서는 아래의 기능들을 어떻게 실현하는가에 대하여 설명한다.

- 추출자
  객체의 내용들을 꺼내서 인쇄한다.
- 삽입자
  C++흐름의 값을 객체안에 삽입한다.
- 입출력조작자
  입출력체계의 동작방법을 변화시킨다. 이것은 보통 정보가 표시되는 형식을 변화시킨다.

22.2.1 추출자

추출자(extractor)는 머리부파일 <iostream>안에 정의된 흐름입출력체계를 사용하여 직접 객체의 내용들을 쓴다. 실례로 이것은 다음과 같은 추출연산자 <<를 다중정의하는것으로 실현되는데 이때 파라메터로서 클라스 Account의 구체례를 가진다.

```
std :: ostream& operator <<( std::ostream &s, Account acc )
{
   s <<std::setiosflags (std ::ios ::fixed);             // x, y 형식
   s<<std ::setprecision (2);                            // 소수점아래 두자리
   s<<std :: setiosflags(std ::ios::showpoint);          // 모든 자리수를 보여 준다
   s << “£”<< acc.account_balance();                     // 세부들을 인쇄
   return s;
}
```

주의: 이 함수는 ostream 객체의 참조를 첫번째 파라메터로 가지며 이 객체의 참조를 그 결과로서 넘겨 준다.
ostream 구체례의 참조는 흐름에 대한 출력관리의 상태정보를 포함하기때문에 객체가 복사될수 없으므로 요구된다.

다음의 코드에서 다중정의된 추출연산자 <<를 사용한다.

```
int main( )
{
    account mike;
     mike.deposit (100.00);
     std ::cout <<”Mike’s account contains “ <<mike <<”\n”;
     return 0;
}
```

해당머리부파일과 함께 콤파일하고 실행하면 결과는 다음과 같다.

```
Mike’s account contains £ 100.00
```

22.2.2 삽입자

삽입자(inserter)는 머리부파일 <iostream>안에 정의된 입출력체계를 사용하여 객체의 내용들을 직접 읽어 낸다. 실례로 이것은 다음과 같은 삽입연산자 >>를 다중정의하는것으로 실현되는데 이때 파라메터로서 클라스 Account의 구체례를 가진다.

```
std :: istream& operator >> (std ::istream &s, Account&  acc)
{
    float money;
    s>>money;                          // 읽기
    acc.deposit (money);               // 객체를 변경
    return s;
}
```

다음의 코드에서 다중정의된 삽입연산자 >>를 사용한다.

```
int main( )
{
   account mike;
   std ::cin >>mike;
   std ::cout <<”Mike’s account contains “ <<mike <<”\n”;
   return  0;
}
```

입력이 50.0 일 때 해당한 머리부파일과 함께 콤파일하고 실행하면 결과는 다음과 같다.

```
Mike’s account contains £ 50.00
```

22.2.3 입출력조작자

입출력조작자(IO manipulator)들은 흐름의 상태를 변화시킨다. 화페량을 쓰는데 적당한 방법으로 float 형구체례를 형식화하는 조작자 money_format 는 다음과 같이 사용된다.

```
int main( )
{
    account mike;
    float gift =100.00;
    std::cout <<”Deposit “ <<money_format << gift <<”\n”;
    mike.deposit(gift);
```

```
    std::cout <<"Mike's account contains" <<mike<<"\n";
    return 0;
}
```

해당머리부파일을 가지고 우의 코드를 콤파일, 실행하면 다음의 결과를 얻는다.

```
Deposit 100.00
Mike's account contains ￡100.00
```

입출력조작자 money_format 는 다음의 두 단계에서 창조된다.

22.2.3.1 조작자창조단계 1

다음의 조작자를 실현하는 함수를 만들어 보자.

```
std :: ostream& money_format (std :: ostream &s)
{
    s<<std :: setiosflags (std :: ios::fixed);          // x.y 형식
    s<<std :: setprecision (2);                          // 소수점아래 두자리
    s<<std :: setiosflags(std :: ios::showpoint);        // 모든 자리수를 보여 준다
    return s;
}
```

주의: 이 함수는 ostream 객체의 참조를 자기의 파라메터로 가지며 이 객체에 대한 참조를 결과로 넘겨 준다.

22.2.3.2 조작자창조단계 2

ostream 객체와 함수 money_format 의 지적자사이에 연산자 <<를 다중정의한다.

```
std :: ostream& operator << (std ::ostream& s,  ostream& (*fun) (std :: ostream&) )
{
    return (*fun) (s);                  // 조작자를 호출
}
```

주의: ostream& (*func) (ostream&)는 ostream 형의 단일파라메터를 가지는 함수의 지적자이며 ostream 형의 객체를 넘겨 준다. 다중정의된 함수는 조작자의 호출을 실현한다.

22.2.3.3 처리

```
std :: cout << "Deposit"<<money_format <<gift << "\n";
```

이 식에서 조작자 money_format 가 실행될 때 다음의 함수들이 호출된다.

- 조작자 money_format 를 호출하는 함수
  std::ostream& operator << (std::ostream& s, std::ostream& (*fun) (ostream&))

가 실행된다.

- 이때 조작자함수
  std::ostream& money_format (std::ostream &s)
  가 조작자들의 기능을 실현한다.

22.2.4 단일파라메터를 가지는 입출력조작자

8개 문자너비를 가지는 마당안에서 화페형식으로 float형으로 형식화하는 조작자 money_format_width(8)은 다음과 같이 사용된다.

```
#include <iomanip>
int main()
{
    account mike;
    float   gift =100.00;
    std :: cout <<” Deposit “ <<money_format_width(8) <<gift <<” \n” ;
    mike.deposit(gift);
    std ::cout <<” Mike’ s account contains “<<mike <<” \n” ;
    return 0;
}
```

주의: 이 처리는 머리부파일 <iomanip>의 포함을 요구한다.

이때 해당한 머리부파일들과 함께 콤파일하고 실행하면 결과는 다음과 같다.

```
Deposit 100.00
Mike’ s account contains ￡100.00
```

22.2.3에서 설명된 파라메터가 없는 조작자들의 처리로서는 단일파라메터를 가지는 조작자들을 직접 취급하기가 어렵다.

입출력조작자 money_format_width는 다음의 두 단계에서 창조된다.

22.2.4.1 단일파라메터를 가지는 조작자창조단계 1

다음의 조작자기능을 실현하기 위한 함수를 만들어 보자.

```
std :: ostream &money_format_width(std ::ostream &s, int field_width)
{
    s <<std ::setiosflags(std::ios::fixed);          // x.y 형식
    s<<std ::setprecision(2);                        // 소수점아래 두자리
    s<<std::setiosflags(std::ios::showpoint);        // 모든 자리수를 보여 준다
    s<<std::setw(field_width);                       // field_width안에서 다음
     return s;
}
```

주의: 두번째 파라메터는 입출력조작자의 첫번째 파라메터이다.

22.2.4.2 단일파라메터를 가지는 조작자창조단계 2

본보기클라스 omanip의 구체례를 돌려 주는 다음의 본보기함수를 정의한다.

```
omanip <int>

money_format_width ( int  field_width)
{
  return  omanip<int>  (money_format_width, field_width);
}
```

주의: 본보기안에서 사용된 형은 조작자의 파라메터형이다.

22.2.4.3 처리

머리부파일 <iomanip>안에 정의된 클라스 omanip(어떤 실현부들은 세부적으로 차이날수 있다.)는 다음과 같이 정의된다.

```
template <class  T>

class  omanip
{
 public:
    omanip ( ostream& (*f) (std::ostream&, T), T  z ) : fun(f), arg(z)
    {
    }

   friend  std::ostream&  operator << (std::ostream &s,  omanip<T>  f)
   {
   }

private:
  std::ostream&  (*fun)  (std::ostream&,  T);
  T  arg;
};
```

클라스 omanip의 구체례가 창조될 때 조작자의 주소와 그의 단일파라메터는 창조된 객체안에 기억된다. 클라스 omanip의 동료함수 ostream& operator << (ostream &s, omanip<T> f)는 조작자를 호출한다.

식

```
std::cout << “Deposit” << money_format_width (8) << gift << “\n”;
```

에서 조작자 money_format_width(8)이 실행될 때 다음의 함수들이 호출된다.

- 클라스 omanip의 구체례를 돌려 주는 함수
  omanip <int> money_format_width (int field_width)
  가 실행된다.

- 조작자의 주소와 파라메터를 기억하는 클라스 omanip의 구축자
- 함수 money_format_width(8)을 호출하는 클라스 omanip의 동료함수
  std::ostream& operator << (std::ostream &s, omanip<T> f)
  가 호출된다.
- 이때 조작자함수
  std::ostream &money_format_width (std::ostream &s, int field_width)
  가 조작자의 기능을 실현한다.

22.3 콤퓨터화된 은행체계(2진자료의 사용)

내부배렬이 아니라 임의로 접근된 디스크파일에서 개별적손님들의 은행구좌에 대한 자료를 기억하기 위해 8.10에서 본 콤퓨터화된 은행체계를 다시 실현해 본다. 이러한 실현방법으로 자료를 영구적인것으로 만들수 있다. 클라스 Bank를 다음에 정의된 책임들을 가지도록 실현한다.

메쏘드	책 임
account_balance	어떤 손님의 구좌에 남아 있는 돈을 돌려 준다.
deposit	지정된 손님의 구좌에 돈을 저금한다.
extend	영구적인 집합체에 새 구좌들을 추가한다.
set_min_balance	손님에 대한 초과출금한계를 설정한다.
statement_summary	구좌의 계산서를 인쇄한다.
last_account_no	마지막구좌번호를 돌려 준다.
withdraw	가능하다면 손님의 구좌에서 출금을 진행한다.

클라스 Bank의 C++명세부는 다음과 같다.

```
#ifndef  CLASS_BANK_SPEC
#ifndef  CLASS_BANK_SPEC
#include  <iostream>
#include  <fstream>

class Bank {
public:
    Bank( char [ ] );
    ~Bank( );
    void extend ( long );                                          //구좌확장
    float account_balance( const int ) const;                      // 잔고
    float withdraw( const int , const float );                     // 출금
    void deposit( const int, const float );                        // 저금
    void statement_balance( const int, const float );              // 최소잔고를 설정
    void statement_summary( std:ostream&, const int ) const;
```

```
    int last_account_no( ) const;                    // 마지막구좌번호
protected:
    void read ( Account& data, long pos );           // 구좌를 읽기
    void write( Account& data, long pos );           // 구좌에 쓰기
private:
    long the_max_customers;                          // 손님의 최대수
    mutable  std::fstream the_customers;             // 파일해체자
   bool the_fs_ok;                                   // 모두 성공
};

#endif
```

주의: fstream형의 성원변수 the_customers는 mutable형으로 표시되므로 const성원함수들은 이 값을 변경할수 있다. 그 리유는 읽기조작만이 수행되여도 흐름이 변경되기때문이다. 이것은 account_balance와 같은 연산을 사용하는 클라스사용자에 대하여 일관성을 표시해 준다.

22.3.1 실현부

클라스 Bank의 실현부에서 손님들의 구좌에 대한 구체례의 기억기영상(image)을 보관하기 위하여 2진파일을 리용한다. 클라스 Bank의 구축자는 2진파일을 열고 파일크기를 결정하여 가지고 있는 구좌개수를 계산한다.

```
#ifndef   CLASS_BANK_IMP
#define   CLASS_BANK_IMP

Bank::Bank(char vol_name[])
{
      the_customers.open( vol_name,std:ios::binary | std:ios::in  | std:ios::out );
      the_fs_ok  =the_customers.fail( ) ==0;
      if (!the_fs_ok)
     {
        throw std::runtime_error(“Can not open bank accounts file”);
      }
      the_customers.seekp( 0L,std:ios::end );
      the_max_customers = ( the_customers.tellg( )) / (sizeof(Account));
}
```

해체자는 열려 진 파일을 닫는다.

```
Bank::~Bank( )
{
    if  (the_fs_ok) the_customers.close( );
}
```

주의: 파일은 읽기,쓰기를 위해서 열게 된다.
파일이 존재하지 않으면 길이 0으로 만들어 진다.

메쏘드 extend는 파일끝에 새로운 손님들의 구좌를 추가한다.

```
void Bank ::extend( long size )
{
    Account new_account;
     for ( long pos=0; pos<size; pos++ )
     {
         the_customers.seekp ( (the_max_customers+pos) * sizeof(Account) );
         the_customers.write((char*) &new_account, sizeof (Account));
      }
      the_max_customers = the_max_customers+size;
}
```

이 구좌들은 성원함수 last_account_no에 의하여 돌려 지는 구좌번호와 함께 매 구좌의 번호들까지 가지고 있다.

```
int Blank::last_account_no( ) const
{
    return the_max_customers-1;
}
```

성원함수 account_balance의 실현부에서 함수 read는 이전에 보관된 account객체의 상태를 읽고 객체 customer에 다시 기억한다. 이때 기억기안에 재기억된 customer객체에로 통보문 account_balance가 한번 보내여 진다.

```
float Bank ::account_balance( const int client ) const
{      Account customer;                          // Account의 구체례
      read (customer,client);                     // 보관된 상태에 덧쓰기
      return customer.account_balance( );
}
```

withdraw의 실현부는 우와 비슷한 방법을 쓰는데 객체상태가 변하므로 이것을 디스크에 예비로 보관하는 처리를 진행한다. 메쏘드 write는 디스크에 객체 customer의 상태를 쓰는 방법으로 이 기능을 수행한다.

```
float Bank ::withdraw( const int client, const float money )
{
      Account  customer;                                // Account의 구체례
      read( customer, client );                         // 보관된 상태에 덧쓰기
      float  get =  customer. withdraw( money );
      write( customer, client );                        // 디스크에 상태를 보관
      return  get;
}
```

성원함수 deposit와 set_min_balance, statement_summary도 류사한 다음의 방법으로 실현된다.

```
void Bank ::deposit ( const int client, const float money )
{
    Account customer;                  // Account의 구체례
    read( customer,client );           // 보관된 상태에 덧쓰기
    customer.deposit( money );
    write( customer,client );          // 디스크에 상태를 보관
}

void Bank::set_min_balance( const int client, const float money )
{
    Account customer;                  // Account의 구체례
    read( customer, client );          // 보관된 상태에 덧쓰기
    customer.set_min_balance( money );
    write( customer, client );         // 디스크에 상태를 보관
}

void Bank::statement_summary( std:ostream& s, const int client ) const
{
    Account customer;                  // Account의 구체례
    read( customer, client );          // 보관된 상태에 덧쓰기
    s << "Bank statement summary-";
    s << " account number " << client << "\n";
    s << " £ " << customer.account_balance( );
    s << "on deposit " << "\n";
}
```

함수 read 와 write는 디스크파일에 대한 실제적인 입출력을 진행한다. 여기서 문제점은 읽기, 쓰기대면부가 문자들의 배렬지적자를 요구한다는것이다. 이것은 문자들의 배렬지적자를 Account객체의 지적자로 강제형변환하여 쉽게 해결할수 있다.

```
void Bank::read( Account& data, long pos )
{
    char *buf  = ( char* ) &data;                          // char형으로
    if( !the_fs_ok )
        throw std::runtime_error( " File not open " );     // 내부오유
    the_customers.seekg( pos * sizeof( Account ) );       // 위치를 탐색
    the_customers.read( buf, sizeof(Account));            // 읽기
    if( the_customers.gcount( ) != sizeof( Account ) )
        throw std::runtime_error( " Failed to read record " );  // 읽기실패
```

```
}
void Bank ::write ( Account& data, long pos )
{
    char buf  = ( char* )  &data;                              // char형으로
    if ( ! The_fs_ok )
        throw std::runtime_error( " File not open " );         // 내부오유
    the_customers.seekp( pos * sizeof( Account ) );            // 위치를 탐색
    the_customers.write( buf, sizeof( Account ) );             // 쓰기
    if( the_customers.gcount( ) != sizeof( Account ) )
        throw std :: runtime_error( " Failed to write record " );  // 쓰기실패
}
#endif
```

객체의 기억기영상에 대한 읽기, 쓰기처리와 관련된 경고는 다음의 경우에만 일어 난다.

- 객체가 지적자들을 포함하지 않는다.
- 읽기, 쓰기를 진행하는 기계들의 동작방식이 호환성을 가진다. 실례로 float형의 표현은 동일하다.

22.3.2 종합서술

우의 클라스들과 해당포함지령들을 사용하여 실행되는 개별적프로그람들은 다음과 같다. 첫번째 프로그람은 번호 0-5의 6개 구좌를 가지는 은행을 설정하고 5번 구좌에 50.00파운드를 저금한다.

```
int main ( )
{
    Bank piggy ( " piggy.bnk " );                        // 아주 작은 은행
    piggy.extend( 6L );
    int customer = piggy.last_account_no( );             // 마지막손님
    piggy.deposit( customer, 50.00 );
    return 0;
}
```

두번째 프로그람은 마지막구좌에서 20 파운드를 출금한다.

```
int main( )
{
```

```
    Bank  piggy ( “ piggy.bnk ” );                    // 아주 작은 은행
    float  obtained;
    int  customer = piggy.last_account_no( );

    piggy.statement_summary( std :: cout, customer );

    std::cout << “\n” << “ Transaction withdraw ￡ 20.00 ” << “\n”;
    obtained = piggy.withdraw( customer, 20.00 );
    std::cout << “ piggy Bank gives ￡ ” << obtained << “ \n ”;
    piggy.statement_summary( std::cout, customer );
    return 0;
}
```

두번째 프로그램의 실행결과는 다음과 같다.

```
Bank statement summary - account number 5
￡50 on deposit

Transaction withdraw ￡20.00
piggy Bank gives ￡20
Bank statement  summary - account number 5
￡30 on deposit
```

세번째 프로그램은 마지막구좌에 질문한다.

```
int main ( )
{
    Bank piggy ( “ piggy.bnk ” );                              // 아주 작은 은행
    int  customer = piggy.last_account_no( );                  // 마지막손님
    piggy.statement_summary( std ::cout, customer  );          // 계산서
    return 0;
}
```

세번째 프로그램의 실행결과는 다음과 같다.

```
Bank statement summary - account number  5
￡30 on deposit
```

22.4 자체평가

- 삽입자와 추출자란 무엇인가?
- 다음의것이 어떻게 인쇄될수 있는가?

객 체	형 식
float temperature	소수점아래 3자리를 가지는 8개 문자너비의 마당안에서
int number	8개 문자너비의 마당안에서
int pattern	2진비트패턴으로

22.5 련 습

- 메쏘드 withdraw가 다음의 방법에서 실현될수 있게 이 장에서 서술된 Bank클라스를 다시 쓰시오.

```
float Bank ::withdraw ( const int client, const float money )
{
    Account  customer;
    the_customers >> seek( client ) >> customer;
    float  get = customer.withdraw ( money );
    the_customers << seek( client ) << customer;
    return get;
}
```

23 깊은 복사와 얕은 복사

이 장에서는 여러 객체들사이에 공유된 기억기의 참조들을 계수하는 방법에 대하여 서술한다. 이전 방법으로 공유기억기를 사용할 때는 객체기억기의 깊은 복사(deep copy)와 얕은 복사(shallow copy)가 진행된다. 객체기억기의 얕은 복사를 리용하면 기억기를 더 능률적으로 사용할수 있고 실행시간을 단축할수 있다. 이 처리의 실현부는 값주기연산자와 복사구축자의 다중정의를 포함한다.

23.1 서술자

자료구조체에 대한 지적자를 흔히 서술자(descriptor)라고 한다. 자료구조체를 복사할 때는 일반적으로 자료구조전부를 복사하지 않고 서술자만을 복사한다. 그러나 이러한 복사는 두 객체가 같은 기억기를 공유하게 한다. 이것은 기초를 이루는 기억기가 변경되지 않는 한 안전하다. 만일 기억기를 변경하려면 기초를 이루는 자료구조의 복사를 먼저 진행하여야 한다.

객체들의 기억기를 표현하는데서 서술자를 사용하여 실현부를 효과적으로 구성할수 있지만 이것은 보다 복잡한 코드를 요구한다. 그러나 결과의 복잡성은 그 클라스기구에 의해 제공된 교갑화에 의하여 객체사용자에게 숨겨 진다.

주의: 다음의 실례에서 서술자리용에 대하여 설명한다. 그러나 이렇게 사소한 자료인 경우에는 일반적으로 서술자기구가 쓰이지 않는다.

사람번호는 그림 23-1에서와 같이 번호서술자에 의해 서술될수 있다.

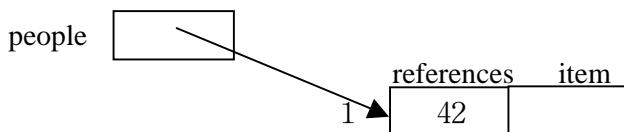



그림 23-1. 번호서술자의 표현

서술자 people은 다음의 2개의 값을 포함하는 서술항목에 대한 지적자이다.

- 이 항목에 대한 참조계수
- 번호의 물리적인 기억기

참조계수를 가진 서술자를 사용하여 여러 객체들은 같은 자료구조체를 공유할수 있다. 실례로 다음의 코드에서

```
Number  people, computing;

computing = 42;
people = computing;
```

번호 42의 기억기는 두 객체 people과 computing사이에 공유된다.

우의 코드를 실행한후 people과 computing을 위한 콤퓨터내부기억기를 그림 23-2에서 보여 준다.

computing

reference item

2 42

people

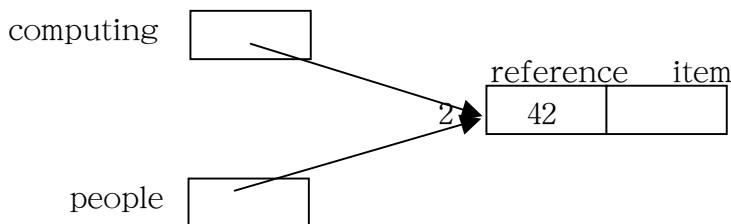



그림 23-2. 같은 기억기를 공유하는 두 객체

이것은 기억된 객체가 큰 경우 기억기를 상당히 줄이고 실행시간을 단축하게 한다. 그렇지만 여러개의 변수가 같은 자료구역을 참조할수 있으므로 클라스 Number에 대한 코드를 작성할 때 주의해야 한다.

클라스 Number에 대한 코드를 작성할 때 고려할 기본항목들은 다음과 같다.

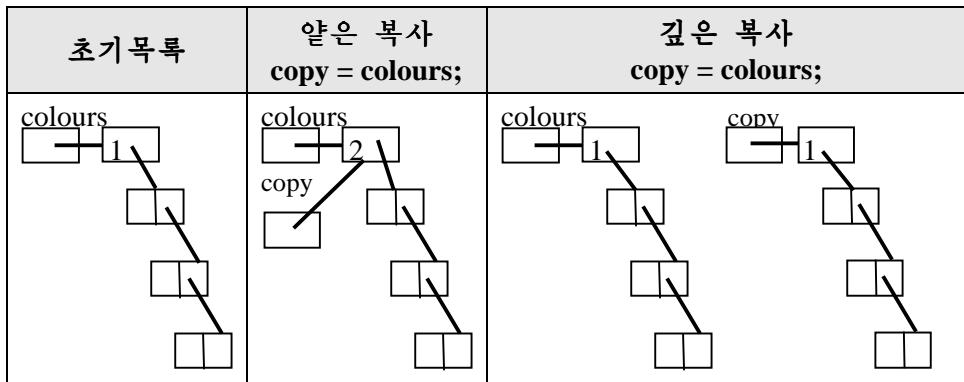
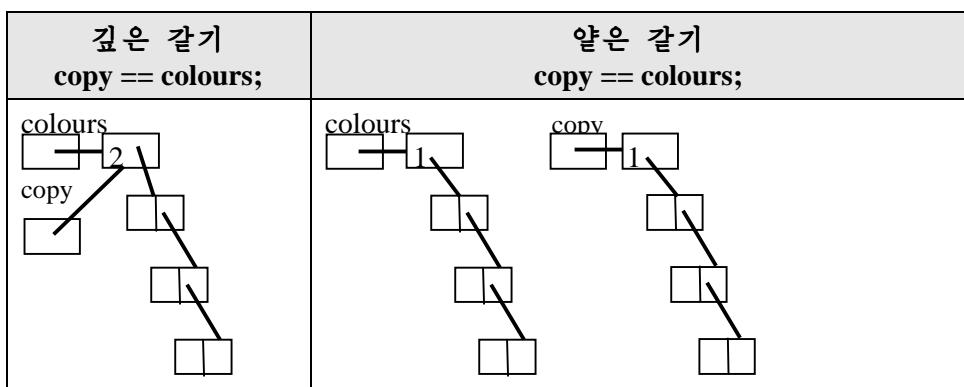
- 자료항목복사에는 두개의 형태가 있다.
  - o 깊은 복사 - 항목에 대한 기억기가 물리적으로 중복된다.
  - o 얕은 복사 - 같은 물리적기억기복사를 공유하는 2개 혹은 그이상의 객체들을 지적하기 위해 참조계수만이 증가된다.
- 값주기연산자는 다중정의되여야 한다.
  변수가 덧쓰기될 때 그의 이전 내용들을 없애는 방법으로 체계를 다시 해방시켜 주어야 한다.
- 복사구축자가 제공되여야 한다.
  이것은 함수에 값을 넘기는것과 같은 암시적인 값주기들을 빨리 처리하기 위해서이다. 복사구축자는 다음의 상태에서 호출된다.
  - o 파라메터는 값으로 넘겨 진다.
    복사구축자는 파라메터의 실지값을 초기화하는데 사용된다.
  - o 함수들은 식을 돌려 준다.
    복사구축자는 식을 돌려 주는 객체의 림시구체레로 변환하는데 사용된다.
- 클라스 Number의 해체자는 참조계수가 1일 때에만 기억기를 해방할수 있다.

이 처리는 아래의 color련결목록을 보면서 설명하기로 한다.

- 값주기

초기목록	얕은 복사 **copy = colours;**	깊은 복사 **copy = colours;**
colours 1	colours 2 copy	colours 1 copy 1

- 갈기

깊은 갈기 **copy == colours;**	얕은 갈기 **copy == colours;**
colours 2 copy	colours 1 copy 1

23.2 깊은 복사와 얕은 복사를 수행하는 클라스

본보기클라스 RC는 어떤 C++의 형이나 클라스에 대하여 깊은 복사와 얕은 복사를 수행하는 참조계수기구를 실현한다. 실례로 다음의 선언

```
typedef  RC <double> Real;
Real  number1, number2;
```

은 double형객체의 참조계수서술자로서 Real을 정의한다. 파라메터화된 RC형에 그의 실제파라메터(이경우는 double)형의 객체를 넘겨 주는 변환함수를 정의함으로써 number1과 number2는 일반적인 방법대로 산수식에서 사용될수 있다.

이 기구는 같은 형의 많은 구체례들이 같은 값을 가질 때 사용될수 있다. 실례로 어떤 걸면의 격자점높이는 다음의 참조계수 long double로서 서술될수 있다.

```
typedef  RC <long double> Real;

void main()
{
  const int  SIZE = 10;
  Real  heights[SIZE][SIZE];
```

```
  for( int i=0; i<SIZE; i++ )
    for( int j=0; j<SIZE; j++ )
      heights[i][j] = Real(0.0);

  return 0;
}
```

우에서와 같이 모든 값이 0일 때는 높이의 SIZE * SIZE지적자들을 가지는 long double값 0.0의 하나의 물리적구체례만이 있다.

클라스 RC의 명세부는 다음과 같다.

```
#ifndef  CLASS_REFERENCE_COUNT_SPEC
#define  CLASS_REFERENCE_COUNT_SPEC

template <class Type>
class RC {
public:
  RC();
  RC( const Type );
  RC( const RC<Type>& );                          // 복사구축자
  ~RC();                                          // 해체자

  RC& operator = ( const RC<Type>& );             // 얕은 복사
  RC& operator<<= ( const RC<Type>& );            // 깊은 복사
  operator Type&( ) const;                        // 변환연산자
  void describe( ) const;                         // 항목을 서술
protected:
  Type& stored( ) const;                          // 내용
  int rdfs( ) const;                              // 참조회수
  void change( Type );                            // 변경
  void spilt( );                     // 가르기(한개 객체를 두개로 가르기)
  void release_storage( );                        // 항목의 기억기를 돌려 준다
  void fail( const char[ ] ) const;               // 포기
private:
  struct  Desc_RC {
    Type  item;                                   // 기억된 자료객체
    int  references;                              // 객체에 대한 참조
  };
  Desc_RC  *the_p_desc;                           // 번호서술자의 지적자
};
#endif
```

RC에 대한 파라메터가 없는 구축자는 그 클라스에 대한 파라메터화된 형으로서 리용된 형의 정의되지 않은 구체레서술자를 창조한다. 만일 이것이 클라스항목이면 값초기화를 위해 호출될수 있는 파라메터가 없는 구축자를 가져야 한다.

```
#ifndef  CLASS_REFERENCE_COUNT_IMP
#define  CLASS_REFERENCE_COUNT_IMP

#include <string>
#include <strstream>
#include <stdexcept>

template <class Type>
RC<Type>::RC()
{
    // 경고
    // 이것은 기억기부족으로 실패할수 있다
    // 실패하는 경우 례외 bad_alloc가 내보내진다

  the_p_desc = new  Desc_RC[1];

  the_p_desc->references = 1;
}
```

다음의 구축자는 파라메터화된 형의 구체례를 자기의 파라메터로 가지는데 이것은 서술자를 초기화하는데 사용된다.

```
template <class Type>
RC<Type> ::RC( const Type value )
{
   // 경고
   // 이것은 기억기부족으로 실패할수 있다
   // 실패하는 경우 례외 bad_alloc가 내보내 진다

  the_p_desc = new Desc_RC[1];

  the_p_desc->item = value;
  the_p_desc->references = 1;
}
```

해체자는 참조계수가 1이라면 기억기를 해방하기만 하는 함수 release_storage를 호출한다.

```
template <class Type>
RC<Type>::~RC( )
{
  release_storage();
}
```

복사구축자는 암시적인 값주기가 있을 때 호출된다. 실례로 실제파라메터가 값에의해 넘겨 질 때 복사구축자가 호출된다. 그 목적객체가 초기화되지 않았다면 그 객체에 대한 기억기도 해방되지 않는다. 지적자는 이미 할당되고 참조회수는 이 객체의 림시참조를 반영하여 하나 증가된다.

```
template <class Type>
RC<Type>::RC ( const RC<Type>& copy )
{
  the_p_desc = copy.the_p_desc;            // 서술자의 지적자를 복사
  (the_p_desc->references) ++;             // 참조를 증가
}
```

다중정의연산자 =(얕은 복사에 사용된다.)는 원천객체의 참조계수를 증가시키고 목적객체에 대한 어떤 목적기억기를 해방하며 그때 그 지적자를 서술자로부터 목적서술자에로 지적자로 값주기한다.

```
template <class Type>
RC<Type>& RC<Type>::operator= ( const RC<Type>& from )
{
  if ( the_p_desc !=from.the_p_desc )            // a=a가 아니다
  {
    (from.the_p_desc->references) ++;            // RC복사
    release_storage();                           // 겹쳐 쓴 RC
    the_p_desc = from.the_p_desc;                // 복사를 진행
  }
  return *this;
}
```

주의: 이러한 연산들의 순서는 사용자가 아래와 같이 쓸수 있으므로 중요하다.
object = object;// 목적과 원천은 같은 항목이다.

다중정의연산자 <<=(깊은 복사에 사용된다.)는 객체기억기의 새로운 구체례를 만들며 여기에 원천객체를 값주기한다. 목적객체의 참조계수는 감소하며 그의 지적자는 새로운 객체서술자의 지적자에 의해 덧쓰기된다.

```
template <class Type>
RC<Type>& RC<Type>::operator <<= ( const RC<Type>& from )  //깊은 복사
{
  Desc_RC *p_desc;
   //경고
   //이것은 기억기부족으로 실패할수 있다
   //실패하는 경우 례외 bad_alloc가 전파된다
  p_desc = new Desc_RC[1];
  p_desc->item = from.the_p_desc->item;
  p_desc->references = 1;
  release_storage();                  // 목적객체의 기억기를 해방한다
  the_p_desc = p_desc;                //목적객체를 설정
  return  *this;
}
```

다음의 세 성원함수들은 서술자안의 구성요소들에 접근하는데 리용된다. 이러한 수법으로 RC클라스의 파생클라스가 그의 비공개부자료성원들에 접근할수 있다.

```
template <class Type>
Type& RC<Type>::stored() const
{
  return the_p_desc->item;                              // 기억된 항목
}
template <class Type>
int RC<Type>::refs() const
{
  return the_p_desc->references;                        // 참조회수
}
template <class Type>
void RC<Type>::change( Type new_value )
{
  the_p_desc->item = new_value;                         // 항목을 변경
}
```

보호부성원함수 split는 참조계수항목을 2개의 개별적인 객체들로 가르기 위해서 사용된다. 한개객체는 그 객체에 대한 참조를 1번만 하며 그밖의 객체들은 그 객체에 대한 참조를 1번이상 한다.

이것은 그 객체에 대한 참조를 2번이상하는 객체에 대하여 변이자성원함수를 실현할 때 사용된다.

```
template  <class Type>
void  RC<Type>::split()
{
   if  (refs( ) >=2 )
   {
     Desc_RC  *p_desc;
     // 경고
     // 이것은 기억기부족으로 실패할수 있다
     // 실패하는 경우 례외 bad_alloc가 전파된다
     p_desc = new Desc_RC[1];
     p_desc-> item = the_p_desc->item;             // 깊은 복사
     p_desc-> references = 1;
     the_p_desc-> references --;                   // 참조회수를 하나 감소
     the_p_desc = p_desc;                          // 한번 참조
   }
}
```

성원함수 release_storage는 객체의 참조계수를 감소시키며 그것이 0이 되는 경우에는 그 객체의 기억기를 물리적으로 해방한다.

```
template <class Type>
void  RC<Type>::release_storage( )
{
   if ( -- (the_p_desc->references) == 0 ) {
      delete[ ] the_p_desc;                        // 그의 서술자
   }
}
```

성원함수 fail은 례외 runtime_error를 발생시킨다.

```
template <class Type>
void RC<Type>::fail( const  char message[] ) const
{
    char storage[120];                                   // 통보문
    ostrstream text (storage,120);                       // 흐름으로
    text<<”RC:”<<message<<’\0’;
    throw std::runtime_error( std::string(storage) );    // 례외
}
```

변환연산자는 기억된 객체구체례의 참조를 돌려 준다. 참조는 기억된 객체가 변경될수 있도록 돌려 진다.

```
template <class Type>
RC<Type>::operator Type& () const
{
   return the_p_desc->item;                 // 기억된 항목
}
#endif
```

23.3 리용실례

다음의 프로그람은 사과와 배의 알수를 기억하는 옹근수의 참조계수를 리용한다. 이 프로그람에서 볼수 있는것처럼 Int는 코드작성자가 《볼수 없는》 참조계수이다.

```
 typedef  RC<int> Int;
int main( )
{
   int total_fruit =0, apples=20, pears=30;
   std::cout<<  “total_fruit = apples + pears” <<  “\n”;
   total_fruit = apples + pears;
   std::cout <<  “total_fruit = ” << ( int ) total_fruit;
   std::cout<<  “apples = ”      << ( int ) apples;
```

```
    std::cout<<  "pears = "     << ( int ) pears <<  "\n" ;
    return 0;
}
```

우의 프로그람을 콤파일하고 실행하면 다음의 코드가 얻어 진다.

```
total_fruit = 50  apples = 20  pears = 30
```

total_fruit = apple + pears; 의 평가에서는 다음의 연산들이 진행된다.

apples + pears;

- int를 넘겨 주는 apples와 pears에 적용된 클라스 RC의 변환연산자
  - apples는 int로서 20을 넘겨 준다.
  - 배는 int로서 30을 넘겨 준다.

  더하기연산자 '+'(표준C++옹근수연산자)는 50을 넘겨 준다.

total_fruit = 50(20+30의 결과)

- 구축자 RC<int> (50)은

  림시기억기를 넘겨 준다.

  값주기연산자 '='는 림시기억기를 준다.
  - total_fruit에 대하여 함수 release_storage를 호출한다.
  - 객체지적자의 값주기를 진행한다.

    *this를 돌려 준다.

림시기억기(apples+pears의 결과)는 유효범위밖으로 벗어 난다.

- ~RC <int> ()

  그러나 참조계수가 0이 아니므로 50에 대한 기억기를 해방하지 못한다.

주의: 산수적연산은 간단하지만 실행되는 코드량은 상당히 많다. 우에서 본 코드경로를 줄이기 위해 값주기의 전문화를 리용한다. 실례로 함수 RC& operator = (const Type&);를 들수 있다.

23.4 객체에 대한 값주기방지

클라스의 비공개부 혹은 보호부성원들에 대한 다중정의연산자 =와 복사구축자를 정의하여 객체사용자가 클라스에서 객체를 복사하는것을 방지한다. 또한 콤파일할 때 클라스사용자가 연산을 볼수 없다는 오유통보문을 내보낸다. 실례로 5.3.1에 서술된 클라스 Account는 다음과 같이 정의될수 있다.

```
#ifndef  CLASS_ACCOUNT_SPEC
#define  CLASS_ACCOUNT_SPEC

class Account {
public:
  Account( const float=0.00, const float=0.00 );
  ~Account();
  float  account_balance() const;                  // 잔고를 돌려 준다
```

```
  float  withdraw( const float );                  // 구좌에서 출금
  void  deposit( const float );                    // 구좌에 저금
  void  set_min_balance( const float );           // 최소잔고를 설정
protected:
  Account( Account& );                             // 복사구축자
  Account& operator = ( Account );                // 값주기
private:
  struct Account_data;                            // 림시선언
  Account_data*  the_storage:                     // 객체에 대한 기억기의 지적자
};

#endif
```

주의: 값주기와 복사구축자의 실현부는 빈 동작을 수행한다.

새로 정의된 클라스 Account를 사용하여 은행구좌안의 돈을 복사하는 아래의 정확치 못한 코드는 콤파일시에 방지된다.

```
int  main()
{
  Account  mike, corinna;
  mike.deposit( 100.00 );
  corinna = mike;                     // 콤파일할 때 오유통보문이 발생된다
  return 0;
}
```

23.5 자체평가

- 객체의 깊은 복사와 얕은 복사사이에는 어떤 차이가 있는가?
- 객체의 깊은 비교와 얕은 비교사이에는 어떤 차이가 있는가?
- 객체의 얕은 복사를 언제 사용할 때 문제가 생기는가?
- 객체의 깊은 복사를 언제 사용할 때 문제가 생기는가?

23.6 련 습

- 클라스 Person_RC
  8.13에서 서술된 클라스 Person에 참조계수를 실현하는 클라스 Person_RC를 실현하시오. 이 새로운 클라스의 사용자는 깊은 복사와 얕은 복사가 진행되는것을 몰라야 한다.
- 클라스 Vector
  클라스의 구체례가 정확히 복사될수 있도록 20.2에서 본 클라스 Vector를 다시 실현해 보라.

24 지적자와 범용알고리듬

이 장에서는 표준본보기서고안의 알고리듬실현에 사용되는 지적자들의 리용법에 대하여 서술한다. STL(standard Templete Library: 표준본보기서고)은 자료를 조작하고 처리하는 표준방법을 정의하며 모든 ANSI C++콤파일러들에서 리용할수 있다.

24.1 범용알고리듬

앞으로 서술하게 될 범용알고리듬(generic algorithm)들은 STL(표준본보기서고)안에 들어 있다. 유용한 함수들이 들어 있는 이 서고는 기억기호출이 지적자위주로 되여 있다. 그 리유는 STL의 알고리듬들이 클라스의 구체례뿐만아니라 내장된 형의 구체례를 가지고 작업할수 있게 하기 위해서이다.

그리고 STL은 기억된 항목들의 관리를 쉽게 해주는 어떤 범위의 용기들을 실현한다. STL의 구조에 대하여서는 25장에서 구체적으로 서술한다. STL은 원래 휼레트팩카드(Hewlett-Packard :HP)회사에서 설계하였는데 인터네트상에서 무료로 리용할수 있다.

C++의 ANSI표준에서는 이러한 범용알고리듬들이 이름공간 std안에서 실현된다. 대역이름공간을 주어 오유를 발생할수 있는 지령 using namespace std;를 사용하는 것보다는 유효범위해결연산자를 리용하면 매 알고리듬을 보기할수 있으므로 이름공간 std에서 알고리듬을 개별적으로 선택할수 있다. 머리부파일 <algorithm>에는 이러한 범용알고리듬들이 들어 있다.

24.2 범용알고리듬 copy

자료를 한 위치에서 다른 위치로 복사하는 범용알고리듬은 다음의 본보기함수 copy에 의하여 실현된다.

```
template <class I_It, class O_It>
O_It copy(  I_It first, I_It last, O_It too )
{
  while ( first !=last )  *too ++ = *first ++;
  return too;
}
```

주의: 이것은 표준서고부분으로서 이름공간 std에서 실현된다.

알고리듬을 실현할 때 본보기함수 copy의 파라메터들은 다음과 같다.

파라메터	설　명
first	복사되여야 할 집합의 첫번째 요소에 대한 입력지적자
last	복사되여야 할 집합의 마지막요소의 다음요소에 대한 입력지적자
too	자료가 이동되여 갈 구역에 대한 출력지적자

주의: 복사되여야 할 기억령역을 지적하기 위해 리용되는 지적자반복자(pointer iterator)는 첫번째 요소와 앞으로 복사되여야 할 기억령역의 다음요소를 지적한다.

24.2.1 종합서술

우의 범용알고리듬 copy를 리용하여 옹근수배렬 numbers의 요소들을 옹근수배렬 copy_of에로 복사하는 코드는 다음과 같다.

```
# include <algorithm>

int main()
{
  int  number[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
  const int  BEGIN = 0;                                   // 첨수
  const int  SIZE = sizeof (number) / sizeof (int);       // 요소들이 없다
  const int  END = SIZE;                                  // 첨수

  int  copy_of[SIZE];

  std::copy ( &number[BEGIN]), &number[END], &copy_of[0] );

  return 0;
}
```

주의: &numbers[BEGIN]은 복사되여야 할 집합의 시작을 가리키는 지적자를 돌려 준다.
&numbers[END]는 복사되는 집합의 마지막요소의 다음요소에 대한 지적자를 돌려 준다.
알고리듬들은 머리부파일 <algorithm>에 포함되여 있다.

24.3 범용알고리듬 for_each

본보기함수 for_each는 집합의 매 요소에 어떤 함수를 적용한다. 본보기함수 for_each의 파라메터들은 다음과 같다.

파라메터	설　명
first	집합의 시작위치에 대한 입력지적자
last	집합의 마지막요소의 다음요소에 대한 입력지적자
f	first부터 last-1까지의 범위안의 매 요소에 적용할 함수로서 그것은 void f (Type&)라고 서술된다.

본보기함수 for_each의 실현부는 다음과 같다.

```
template <class IO_It, class Unary_function>
Unary_function for_each( IO_It  first,  IO_It  last,  Unary_function  f )
{
 while ( first != last )                         // 매 요소에 대해
 {
   f( *first ++ );                               // 매 요소에 함수 f를 적용
 }
 return  f;
}
```

24.3.1 종합서술

우의 알고리듬 for_each를 리용하여 옹근수배렬 numbers에 들어 있는 매개 요소들의 내용을 인쇄하는 코드는 다음과 같다.

```
# include <algorithm>
void print_int ( int& val )
{
   std::cout << val << “ ” ;
}
int main( )
{
    int number[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
    const int  BEGIN = 0;                                    // 첨수
    const int  SIZE = sizeof (numbers) / sizeof (int);       // 요소들이 없다
    const int  END = SIZE;                                   // 첨수

    std::for_each( &numbers[BEGIN]), &numbers[END], print_int );
    std::cout << “ \n” ;
    return 0;
}
```

알맞는 머리부파일을 포함시켜 콤파일하면 그 결과는 다음과 같다.

```
10  9  8  7   6  5  4  3  2  1
```

24.3.2 함수객체(범용인쇄를 실현하기)

우에서 본 방법의 제한성은 인쇄될 개개의 자료형에 해당한 함수 print의 구체례가 있어야 한다는것이다. 우의 실례에서 함수 print_int는 옹근수값만을 인쇄한다.

함수객체는 함수호출연산자를 다중정의하는 클라스의 구체례이다. 따라서 함수객체는 일반적으로 함수처럼 사용된다. 실례로 print가 함수객체라면 print( )는 그 함수기

능을 실현하는 메쏘드. operator( )를 호출한다. 함수객체를 가지는 클라스는 범용클라스로 될수 있으므로 해당함수객체의 구체례는 구체례제시(instantiation)되여 해당자료형을 처리하여야 한다. 다음의것은 형 Type의 인수를 인쇄하는 함수객체 print<Type>()이다.

```
template <class Arg1, class Result>
class unary_function
{
public:
    typedef Arg1  first_arg_type;
    typedef Result result_type;
};

template <class Type>
class print : unary_function<Type, void>
{
public:
    void operator ( ) (const  Type& x) const
    {
       std::cout <<x <<  " " ;
    }
};
```

주의: 클라스 unary_function은 모든 함수객체들을 위한 일반기초클라스로 리용된다. 특히 클라스는 함수객체안에서 사용된 형들로 이름 지어 진다.

옹근수배렬 numbers의 내용을 인쇄하기 위해 사용된 함수객체 print<int>()는 다음과 같다.

```
int main ( )
{
    int  number[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
    const int  BEGIN = 0;                                      // 첨수
    const int  SIZE     = sizeof(numbers) / sizeof(int);       // 요소들이 없다
    const int  END      = SIZE;                                // 첨수

    std::for_each( &number[BEGIN], &numbers[END], print<int>( ) );
    std::cout <<"\n";
    return  0;
}
```

주의: 클라스상수 print<int>()는 범용클라스 print<int>의 구체례이다.

24.3.3 함수객체의 작용방법

범용알고리듬 for_each에서 세번째 파라메터는 집합의 매 요소에 적용되는 함수로 리용된다.

호출

```
for_each(&numbers[START], &number[END], print<int>( ) );
```

에서 세번째 실제파라메터 print<int>( )는 범용클라스 print의 구체례인데 이것은 집합의 매 요소에 적용된다. 알고리듬 for_each(24.3을 보시오.)에서 함수를 호출하는 행은

```
f(*first++);
```

인데 이것은

```
instance_of_print( *first++ );
```

로 콤파일되며 다음과 같이 실현된다.

```
instance_of_print.operator ( )  ( *first++ );
```

주의: 이것은 클라스 print가 함수호출연산자 ()를 다중정의하였기때문이다.

24.4 정렬법

일부 가장 단순한 정렬(sort)알고리듬들은 거품정렬에 기초하고 있다. 커지는 순서를 가지는 거품정렬에서는 항목들의 쌍이 련속적으로 서로 비교되며 필요하다면 정확히 커지는 순서로 정돈된다. 이 처리의 결과는 더 큰 항목들은 목록의 끝에로 이동하는것이다. 목록을 통과하는 처리는 린접한 항목들을 교체하면서 목록안에 있는 모든 함수들이 정확한 순서로 놓일 때까지 반복된다. 실례로 다음목록의 수자들은 커지는 순서로 분류된다.

20	10	17	18	15	11

거품정렬의 첫번째 통과는 린접한 수자들의 쌍을 비교하고 매쌍을 커지는 순서로 정돈한다. 이것은 아래의 그림 24-1에서 설명된다.

20	10	10	10	10	10
10	20	17	17	17	17
17	17	20	18	18	18
18	18	18	20	15	15
15	15	15	15	20	11
11	11	11	11	11	20

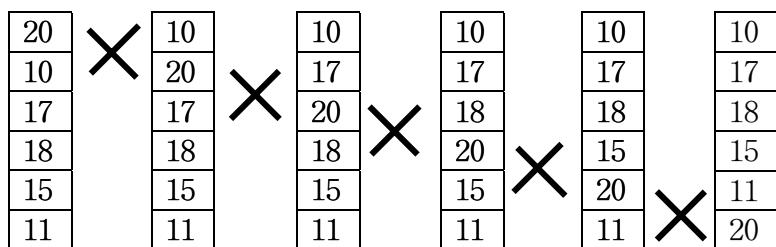



그림 24-1. 거품정렬의 첫번째 통과

매번 수자목록을 통과할 때마다 보다 큰 수자들을 목록의 끝방향으로 이동시키고 보다 작은 수자들을 시작방향으로 이동시킨다. 이때 목록안의 추가적인 보다 큰 수자만이 정확한 위치에 놓이게 된다. 수자목록의 성과적인 통과결과는 아래의 표에서 설명된다.

수목록	설　명
20 10 17 18 15 11	초기목록
10 17 18 15 11 20	첫번째로 목록을 통과한후
10 17 15 11 18 20	두번째로 목록을 통과한후
10 15 11 17 18 20	세번째로 목록을 통과한후
10 11 15 17 18 20	네번째로 목록을 통과한후

이 처리는 목록을 한번 통과하여 자리교체가 일어 나지 않을 때까지 반복된다. 4번째 통과후 목록의 다음통과에서는 자리교체가 일어 나지 않을것이다. 이것은 목록이 커지는 순서로 정렬되였다는것을 의미한다.

24.4.1 효률

이러한 거품정렬의 변종은 효률적이지 못한 알고리듬이며 그 경우에는 커지는 순서로 자료를 재정돈하기 위하여 목록을 n번 통과해야 한다. 여기서 n은 목록의 크기이다. 매 통과마다 n-p번의 비교가 일어 나는데 여기서 p는 통과수이다.

큰 O표식은 알고리듬의 근사적인 차수(order)를 주기 위해 리용되는데 변경된 거품정렬에 대하여 차수(비교회수)는 대략 $O(n^2)$이다. 자료의 량이 작을 때는 이것이 문제로 되지 않지만 n이 클 때는 비교회수가 매우 많아 지며 이로 하여 자료를 정렬하는데 걸리는 시간은 길어 질것이다.

24.5 범용알고리듬 sort

다음의 코드는 자료를 커지는 순서로 정돈하는 거품정렬의 변경된 형식을 사용한다. 매번 자료를 통과한후에 마지막항목은 정확한 순서로 놓인다. 그후의 통과들은 그전의 통과로부터 순서대로 놓인 마지막항목을 무시할수 있다. 범용알고리듬 sort의 STL실현부에서는 훨씬 빠른 정렬알고리듬을 리용한다.

```
template <class IO_It>
void sort ( IO_It first, IO_It last_plus_one )
{
  bool  swaps = true;                          // 정렬되지 않음
  IO_It last = last_plus_one-1;                // 실지 마지막값
   while  (swaps)
```

```
  {
   swaps = false;                                   // 정렬되였다고 가정
   for(IO_It p=first; p!=last; p++)                 // 배렬훑기
   {
     IO_It q = p+1;                                 // q는 다음항목을 가리킨다
     if ( *p > *q);                                 // 쌍 (p, q)가 틀린 순서이면
     {
       swaps = true;                                // 따라서 정렬되지 않음
       swap ( *p, *q);                              // 쌍 (p, q)를 정확한 순서로 놓기
     }
    }
   last--;                                          // 마지막항목은 정확하다
  }
}
```

함수 swap는 본보기함수로 실현된다.

```
template <class Type>
void swap (Type& first, Type& second)
{
  Type tmp = first;                                 // 림시복사
  first = second;                                   // 첫번째 변경
  second = tmp;                                     // 두번째 변경
}
```

24.5.1 종합서술

우의 알고리듬 sort를 리용하여 정확한 순서로 배렬의 수들을 정렬시키는 코드부분은 다음과 같다.

```
const int BEGIN = 0;                                //첨수
const int SIZE    =sizeof(number)/sizeof(int);      //요소들이 없다
const int END     = SIZE;            //첨수
sort (&numbers [START]), &numbers[END] );
```

주의: 정렬되여야 할 항목들의 집합은 첫번째 항목지적자와 마지막항목다음의 지적자에 의하여 서술된다.
STL정렬알고리듬은 std::sort이다.

24.5.2 차수를 가지는 범용알고리듬 sort

다중정의본보기함수 sort에서 세번째 파라메터는 정렬차수(sort order)를 결정하는데 리용된다.

파라메터	설　명
first	집합의 시작에 대한 입력지적자
last	집합의 마지막요소의 다음요소를 지적하는 입력지적자
cmp	2개의 항목이 정확한 순서로 놓여 있으면 참을 돌려 준다.

다중정의본보기함수 sort는 다음과 같이 실현된다.

```
template <class IO_It, class Compare_function>
void sort ( IO_It first, IO_It last_plus_one, Compare_function cmp)
{
 bool swaps = true;                  // 정렬되지 않음
 IO_It last = last_plus_one-1;       // 실지 마지막값
 while ( swaps )
 {
  swaps = false;                     // 정렬되였다고 가정
  for (IO_It p=first; p!=last;  p++) // 배렬훑기
  {
    IO_It  q = p+1;                  // q는 다음항목을 지적한다
    if ( !cmp( *p,*q) )              // 쌍 (p, q)가 틀린 순서이면
    {
      swaps = true;                  // 따라서 정렬되지 않음
       swap( *p, *q);                // 쌍 (p, q)를 정확한 순서로 놓기
     }
   }
    last--;                          // 마지막항목은 정확하다
 }
}
```

주의: 본보기함수 swap의 실현부는 범용알고리듬 sort의 이전 판에서 보여 주었다.

24.5.3 종합서술

클라스 Person은 다음의 메쏘드들을 가진다.

메쏘드	기 능
Person	이름과 키를 가진 사람에 대한 구체례를 구성
name	사람이름을 문자렬로 돌려 준다
height	사람의 키를 cm로 돌려 준다

클라스 Person의 명세부는 다음과 같다.

```
#ifndef  CLASS_ACCOUNT_SPEC
#ifndef  CLASS_ACCOUNT_SPEC
#ifndef  <string>

class Person
 {
public:
  Person ( std::string = “ ” , int = 0);
  std::string name( ) const;                    // 이름
  int   height ( ) const;                       // 키 (cm)
private:
  std::string the_name;                         // 이름
  int         the_height;                       // 키 (cm)
};

#endif
```

클라스 Person의 실현부는 다음과 같다.

```
#ifndef CLASS_ACCOUNT_IMP
#ifndef CLASS_ACCOUNT_IMP

Person::Person ( std::string name, int height)
{
  the_name = name;  the_height = height;        // 세부들을 기억
}
std::string Person::name( ) const
{
  return the_name;                              // 사람의 이름을 돌려 준다
}
int Person::height ( )const
{
  return the_height;                            // 사람의 키를 돌려 준다
}

#endif
```

사람들의 집합을 정렬하는 프로그람을 아래에 보여 준다. 프로그람은 2개의 함수 즉 첫번째 사람의 이름이 두번째 사람의 이름보다 자모순서에서 보다 앞에 있다면 참을 돌려 주는 cmp_name과 첫번째 사람이 두번째 사람보다 더 작으면 참을 돌려 주는 cmp_height를 리용한다.

```
bool cmp_name ( Person first, Person second)
{
     return first.name( ) < second.name( );
```

```
}
bool cmp_height (Person first, Person second)
{
    return first.height( ) < second.height( );
}
```

추출연산자를 재정의하며 클라스 Person의 구체례를 인쇄하는 코드부분은 다음과 같다.

```
ostream& operator << ( ostream& s, const Person& individual)
{
  s<< "[ " << individual.name( ) << "," <<individual.height( ) << "]";
  return s;
}
```

키와 이름순서로 사람들을 정렬하는 프로그람을 아래에 보여 준다.

```
int  main ( )
{
  Person friends [] = {Person("Paul",165), Person("Carol",147),
                    person("Mike",183),Person("Corinna",171)    };
const int NUM_FRIENDS = sizeof(friends)/sizeof(Person);
const int BEFIN = 0;
const int END     = NUM_FRIENDS-1;

std::sort( &friends [BEGIN], &friends [END], cmp_height);
std::cout << "Tallest person is                :"<<friends [LAST] << "\n";
std::sort( &friends [BEGIN], &friends [END], cmp_name);
std::cout << "Highest collating name is      :"<<friends [LAST] << "\n";

return 0;
}
```

주의: 이 코드부분은 실지로 STL sort함수를 리용한다.

그리고 해당한 머리부파일들과 함께 콤파일될 때 다음의 결과를 출력한다.

```
Tallest person is              : [Mike,183]
Highest collating name is  : [Paul,165]
```

24.6 범용알고리듬 find

본보기함수 find는 제공된 객체에 맞는 요소에 대한 지적자를 돌려 준다. 만일 그 어떤 객체도 찾지 못하면 지적자 last가 돌려 지게 된다.

파라메터	설 명
first	집합의 시작을 지적하는 입력지적자
last	집합의 마지막요소의 다음요소를 지적하는 입력지적자
value	탐색하려는 값

본보기함수 find는 다음과 같이 실현된다.

```
template <class I_It, class Type>
I_It find (I_It first, I_It const last, const Type& value)
{
  while ( (first != last) && (*first != value) ) ++first;
  return first;
}
```

24.6.1 종합서술

우의 알고리듬 find를 사용하여 푸른색이 무지개색중의 한가지 색인가를 결정하는 코드부분은 다음과 같다.

```
int main ( )
{
  std::string colours[] = { “Violet”, “Blue”, “Green”
                            “Yellow”, “Orange”, “Red”};
  const int BEGIN = 0;                                        //첨수
  const int SIZE = sizeof ( colours )/sizeof(std::string);    //요소들이 없다
  const int END  = SIZE;                                      //첨수

  std::string*pos;
  pos = std::find ( &colours[BEGIN], &colours [END], “Blue”);
  if ( pos != &colours [END]) )
  {
    std::cout<< *pos << “is a colour in the rainbow”<< “\n”;
  }
  pos = std::find( &colours[BEGIN], &colours[END], “Pink”};
  if ( pos != &colours [END]) )
  {
    std::cout <<*pos<< “ is a colour in the rainbow” << “\n”;
  }
  return 0;
}
```

또한 해당한 머리부파일들과 함께 콤파일, 실행하면 다음의 결과를 출력한다.

```
Blue is a colour in the rainbow
```

24.7 평가기준을 가진 find범용알고리듬

다중정의된 본보기함수 find는 제공된 객체에 맞는 요소에 대한 지적자를 돌려준다. 만일 그 어떤 대상도 찾지 못하게 되면 지적자 last가 돌려 지게 된다.

파라메터	설 명
first	집합의 시작에 대한 입력지적자
last	집합의 마지막요소의 다음요소를 지적하는 입력지적자
pred	찾는 객체를 결정하기 위한 조건

다중정의본보기함수 find는 다음과 같이 실행된다.

```
template <class I_It, class Predicate>
I_It find_if (I_It first, I_It const last, Predicate pred)
{
   while (fist !=last && !pred(*first)) ++first;
   return first;
}
```

24.7.1 종합서술

우의 알고리듬 find를 사용하여 무지개색중에 3개의 문자들을 가지는 색이 있는가를 결정하는 코드부분을 작성하면 아래와 같다. 우선 문자렬이 #문자를 가지는가를 검사하는 함수객체를 만드는 클라스를 작성하면 다음과 같다.

```
class length_is
{
public:
  explicit length_is (int len) : the_length (len)
  {
  }
  bool operator ( ) (const std::string& str) const
  {
    return str.length( ) == the_length;
  }
private:
  const int the_length;                                    //검사길이
};
```

주의: 함수객체 length _is의 구축자는 찾으려는 문자렬의 길이를 기록한다.
함수객체 length _is는 표준서고에 있지 않다.

이것은 무지개색중에서 3개의 문자들을 가지는것이 있는가를 결정하는 다음의 프로그람에서 사용된다.

```
int main ( )
{
  std::string colours[] = { "Violet"," Blue"," Green",
                            "Yellow"," Orange"," Red"};
  const int BEGIN = 0;                                       // 첨수
  const int SIZE = sizeof(colours)/sizeof(std::string);      // 요소들이 없음
  const int END =SIZE;                                       // 첨수
  std::string*pos;
  pos = std::find_if (&colours[BEGIN], &colours[END], length_is(3) );
  if (pos ! = &colours[END] )
  {
   std::cout << "A colour of the rainbow with 3 characters is"
            <<*pos<< " \n";
  }
  return 0;
}
```

또한 해당한 머리부파일들과 함께 콤파일, 실행되면 다음의 결과를 출력한다.

```
A colour of the rainbow with 3 characters is Red
```

24.7.2 작용방법

일반알고리듬 find에서 세번째 파라메터는 집합의 매 요소에 적용되는 함수객체이다. 함수객체는 해당한 집합요소를 찾으면 참을 돌려 준다.

find_if(&colours[BEGIN], &colours [END], length_is(3));

이 호출에서 세번째 실제파라메터 leugth_is(3)은 클라스 length_is의 구체례이다. 클라스구축자는 구체례변수 the_length를 값 3으로 초기화한다. 범용알고리듬 find의 프로그람본체에서 형식파라메터 pred는 행

while(first<last && ! pred (*first)) ++first;

에서 클라스 length_is에 대한 구체례의 값을 취한다. 식 !pred(*first)는

!instance_of_length_is(*first);

로 해석되는데 이것은

!instance_of_length_is.operator()(*first);

로 실현된다.

클라스 length_is에서 함수호출연산자 ()는 파라메터가 그 객체를 구성하였던 값과 같은가를 결정하는 코드에 의하여 다중정의된다. 만일 탐색하려는 객체를 찾지 못하면 반복자 last가 돌려 지게 된다.

24.8 범용알고리듬 transform

범용알고리듬 transform은 원천집합의 매 성원에 어떤 함수를 적용한다. 원천집합의 매 성원에 대한 함수의 적용결과들은 하나의 결과집합에 기억된다. 이 알고리듬에는 2가지 형태가 있는데 하나는 하나의 집합을 하나의 결과집합으로 전환시키는데 사용되는 1진판본이고 다른 하나는 결과집합을 창조하는 2개의 입력집합들로부터 요소들을 대응시키기 위해서 2진함수가 적용된 2진판본이다. 범용알고리듬 transform은 다음과 같이 정의된다.

파라메터	설 명
first1	첫번째 원천집합의 시작을 지적하는 입력반복자
last1	첫번째 원천집합의 마지막요소의 다음요소를 지적하는 입력반복자
first2 △	두번째 원천집합의 시작을 지적하는 입력반복자
result	결과집합의 시작을 지적하는 출력반복자
f	집합을 전환시키는 함수 2원변환: results = f(collection_1st, collection_2^{nd}) 1원변환: results = f(collection_1st)

주의: △은 2진변환범용알고리듬에 대해서만 적용된다는것을 표시한다.

2개의 변환판본들을 아래에 보여 준다.

```
template <class I_It, class O_It, class Unary_function>
O_It transform (I_It first, I_It last, O_It result,
                    Unary_function f)
{
    while (first != last) *result++ = f(*first++);
    return result;
}
```

```
template <class I_It1, class I_It2,   class O_It, class binary_function)
O_It transform (I_It1 first1, I_It1 last1,
```

```
                    I_It2 first2, O_It result,
                    binary_function f)
{
    while (first1 != last1) * result++ = f(*first1++, *first2++);
    return result;
}
```

24.8.1 transform의 리용실례

다음의 실례프로그람은 옹근수형배렬로부터 요소들을 인쇄하는 함수 print_array를 리용한다. 이것은 ostream_iterator의 두번째 파라메터를 가지는 범용알고리듬 copy를 사용하는것으로 실현된다. 요소들이 ostream_iterator의 구체례에 복사되였을 때 그것들은 지정된 출력흐름에 씌여 진다. 이것은 25.3.5에서 구체적으로 설명된다.

```
void print_array(std::string title, int *from, int *too)
{
  std::cout<<title;
  std::copy(from, too,
            std::ostream_iterator<int> (std::cout, " "));
  std::cout << "\n";
}
```

프로그람의 main함수에서 다음의 범용함수객체들이 변환에 리용된다.

함수객체	실　현
std::plus<int>( )	+
std::negate<int>( )	1원 －

주의: 24.9에서 다른 유용한 함수객체들을 보여 준다.

```
int main ( )
{
  const int BEGIN = 0;
  cosnt int END = 10;
  cosnt int SIZE = END;
  int sourcel[SIZE], source2[SIZE],  target[SIZE];

  for ( int i=0; i<SIZE; i++;
  {
     source1[i] = 10+i;
     source2[i] = 20+i;
  }

 print_array("source1 =", &source1[BEGIN], &source1 [END] );
```

```
  print_array( "source2 =" , &source2[BEGIN], &source2 [END] );

  std::cout<< "Evaluate: target  = source1 + source2" << "\n" ;

  std::transform( &source1[BEGIN], &source1 [END],
                  &source2[BEGIN], &source1 [BEGIN], std::plus<int>( ) );

  print_array( "target =  "," , &target[BEGIN], & target[ END], );

  std::cout << "Evaluate: target = -source1" << "\n" ;

  std::transform( &source1[BEGIN], &source1 [END], &target[BEGIN],
                                                        std::negate<int> ( ) );

  print_array( "target =" , &target[BEGIN], &target[END] );

  return 0;
}
```

24.8.2 종합서술

해당한 선언들과 함께 콤파일, 실행되면 다음의 결과를 출력한다.

```
source1 = 10 11 12 13 14 15 16 17 18 19
source2 = 20 21 22 23 24 25 26 27 28 29
Evaluate: target = source1 + source2
target = 30 32 34 36 38 40 42 44 46 48
Evaluate: target = -source1
target = -10 -11 -12 -13 -14 -15 -16 -17 -18 -19
```

24.9 함수객체

다음의 범용함수객체들은 머리부파일 <algorithm>에서 제공된다. 아래의 표에서는 함수객체의 이름만을 보여 준다. 실례로 2개의 int함수들을 함께 곱하는 함수객체는 times<int>()이고 2개의 문자렬을 련결시키는 함수객체는 plus<std::string>()이다.

이름	연산	이름	연산
plus	x + y	minus	x - y
multiplies	x * y	divides	x / y
modulus	x % y	negate	- x

equal_to	x = = y	not_equal_to	x != y
greater	x > y	less	x < y
greater_equal	x >= y	less_equal	x <= y

이름	연산	이름	연산
logical_and	x && y	logical_or	x || y
logical_not	! x		

24.10 범용알고리듬 generate

본보기함수 generate는 색인된 집합의 련속적인 요소들에 발생기함수의 결과를 기억시킨다.

파라메터	설 명
first	입력집합의 시작을 지적하는 입력지적자
last	입력집합의 마지막요소의 다음요소를 지적하는 입력지적자
f	호출될 때마다 잠재적으로 서로 다른 결과를 넘겨 주는 발생기함수

본보기함수 transform은 다음과 같이 실현된다.

```
template <class O_It, class Generator>
void generate(O_It first, O_It last,Generator f)
{
  while (first != last) *first++ = f ( );
}
```

24.10.1 발생기함수

발생기함수는 호출될 때마다 잠재적으로 다른 값들을 돌려 주는 함수이다. 실례로 피보나치수렬의 련속적인 값들을 돌려 주는 본보기발생기 fibonacci는 다음과 같이 정의된다.

```
template <class Type>
class fibonacci : unary_function<Type, void>
{
public:
  fibonacci( )
  {
    the_first = 0; the_second = 1;                   // 창문의 초기값들
  }
  Type operator ( ) ( )
  {
    Type next = the_first + the_second;              // 피보나치렬에서 다음위치
     the_first = the_second;                         // 창문을 아래로 이동
    the_second = next;
```

```
    return the_first;                       // 피보나치렬에서 다음위치를 돌려 준다
  }
private:
  Type the_first;                           // 창문의 첫번째
  Type the_second;                          // 창문의 두번째
};
```

주의: 구축자는 피보나치수렬의 다음항목계산에서 리용되는 창문의 초기값들을 설정한다. 메쏘드 operator()는 피보나치수렬에서 련속적인 항목들을 넘겨 준다.

24.10.2 종합서술

우에서 서술한 알고리듬 generate와 발생기 fibonacci를 리용하여 피보나치수렬의 항목을 가진 배렬 numbers를 배치하는 코드부분은 다음과 같다.

```
int main ( )
{
    int numbers [15];                                   // std::vector
    const int BEGIN  = 0;                               // 첫번째 첨수
    const int SIZE = sizeof(numbers)/sizeof(int);   // 요소가 없다
    const int END    = SIZE;                            // 마지막첨수
    std::generate(&numbers[BEGIN],&numbers[END], fibonacci <int> ( ) );
    std::cout<< "Fibonacci sequence is : ";
    std::fpr_each (&numbers[BEGIN], &numbers[END], print<int>( ) );
    std::cout << "\n";
    return0;
}
```

주의: 함수객체 print<int>()는 24.3.2에서 정의된다.

또한 해당선언들과 함께 콤파일, 실행되면 다음의 결과가 출력된다.

```
Fibonacci sequence is : 1  1  2  3  5  8  13  21  34  55  89  144  233  377  610
```

24.11 함수적응자

함수적응자(function adapter)는 현재 있는 함수객체의 밖에서 새로운 함수를 만들 때 리용된다. 새로운 함수객체들은 함수적응자를 리용하여 현재 있는 함수객체밖에서 만들어 질수 있다.

24.11.1 함수적응자 not1

함수적응자 not1은 현재 있는 함수객체의 반대결과를 넘겨 주는 새로운 함수를 만들 때 리용된다.

함수객체 even은 다음과 같이 정의된다.

```
template <class Type>
class even : public unary_function<Type, bool>
{
public:
  bool operator ( ) (const Type& va1) const
  {
    return va1%2==0;
  }
};
```

우의 코드는 파라메터로서 짝수를 가질 때 참을 넘겨 준다. 실례로 다음의 코드는 참을 넘겨 준다.

```
even<int> ( ) (4)
```

주의: 함수객체 even의 구체레는 다음과 같이 만들어 진다.

```
even<int>( )
```

함수객체에서 함수는 이 객체에 함수호출괄호를 적용한것에 의해서 호출된다.

함수적응자 not1을 거기에 파라메터로 넘겨 지는 함수객체의 반대결과를 넘겨 주는 새로운 함수객체를 넘겨 준다. 실례로 홀수파라메터가 자기에게 넘겨 질 때 참을 넘겨 주는 새로운 함수객체는 다음과 같이 우에서 본 함수객체 even의 밖에서 창조될수 있다.

```
std::not1 ( even<int> ( )  )
```

주의: not1(even<int>)는 함수객체 even<int>()의 반대결과를 넘겨 주는 함수적응자를 만든다.

새로운 이 함수객체는 옹근수값이 홀수인가를 검사하는 다음의 코드부분에서 리용된다.

```
int main ( )
{
  int i = 2;
  std::cout << "The number" << i << " is "
            << (not1 ( even<int>( ) ) ( i ) ? "odd" : "not odd") << " \n";
return 0;
}
```

주의: 함수호출괄호 ()들은 함수를 호출하는데 사용된다.

24.11.2 작용방법

함수적응자 not1의 구축자는 함수객체 even<int>()를 기억한다. 함수호출괄호가 함수적응자에 적용될 때 성원함수 operator()는 기억된 함수객체 event<int>( )의 호출을 요구하며 그 반대결과를 넘겨 준다. 이 처리를 그림 24-2에서 설명한다.

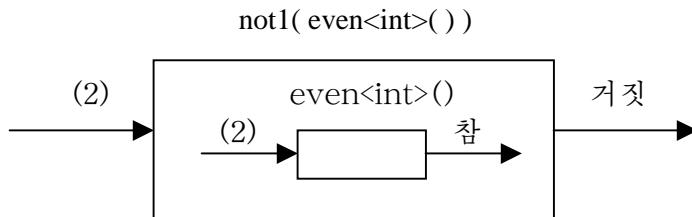



그림 24-2. 함수객체 even의 반대결과를 넘겨 주는 함수적응자

함수적응자 not1은 다음과 같은 2개의 본보기클라스들로서 정의된다.

```
template <class Predicate>
class unary_negate:
public unary_function<typename Predicate::first_arg_type, bool>
{
public:
   explicit unary_negate ( const Predicate& x) : pred(x) ( )
   bool operator ( ) )const first_arg_type& x) const (return !pred(x);)
private:
   Predicate pred;
};
template <class Predicate>
unary_negate<Predicate> not1(const Predicate& pred)
{
   return unary_negate<Predicate> (pred);
}
```

주의: 예약어 typename은 Predicate:: first_arg_type가 하나의 형이라는것을 가리킬 때 쓰인다.

적응자함수는 기억된 항목이 함수객체인 때를 제외하고 일반함수객체와 비슷한 방법으로 작용한다.

24.11.3 종합서술

우에서 서술한 적응자 not1과 함수객체 even을 리용하여 피보나치렬에서 첫번째 짝수와 홀수들을 찾는 코드는 다음과 같다.

```
int main ( )
{
  int number[15];                                    //std::vector
```

```
  const int BEGIN = 0;                                                    //첫번째 첨수
  const int SIZE = sizeof(numbers)/sizeof(int);                           //요소가 없다
  const int END =SIZE;                                                    //마지막첨수
  generate( &numbers[BEGIN], &numbers[END], fibonacci<int> ( ) );
  int *p;
  p= find_if(&numbers[BEGIN], &numbers[END], even<int> ( ) );
  std::cout << "first      even number is " << p << "\n";
  p= find_if(&numbers[BEGIN], &numbers[END], not1 (even<int>( ) ) );
  std::cout << " first not even number is" << *p << "\n";
  return 0;
}
```

실행결과는 다음과 같다.

```
first      even number is 2
first not even number is 1
```

24.11.4 함수적응자 bind2nd와 bind1st

함수적응자 bind2nd는 2개의 파라메터를 가지며 T fun_new(T2 x)로서 새롭게 서술되는 함수를 넘겨 주며 T fun(T1 x, T2 value)를 실현한다.

첫번째 파라메터	T fun(T2 first, T2 second)로서 서술되는 함수
두번째 파라메터	T2형의 값

실례로 단정확도형옹근수파라메터가 100보다 큰가를 검사하는 함수는 bind2nd (greater<int>(),100)과 같이 구성된다.

함수적응자 bind2nd의 구축자는 함수객체 greater<int>()와 옹근수값 100을 기억한다. 그다음 실제파라메터 999가 들어 있는 함수호출괄호가 그 함수적응자에 적용되면 실제파라메터 999와 100을 가진 함수객체 greater<int>()를 불러 내는 메쏘드 operator(999)가 호출된다. 이 처리를 그림 24-3에서 설명한다.

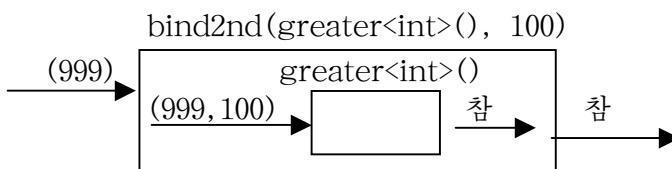



그림 24-3. 파라메터가 100보다 크면 참을 넘겨 주는 함수적응자

함수적응자 bind1st는 2개의 파라메터

첫번째 파라메터	T fun(T2 first, T2 second)로 서술되는 함수
두번째 파라메터	T2형의 값

를 가지며 T fun_new (T2 x)로 새롭게 서술되는 함수를 넘겨 주고 T fun(T2 value, T2 x)를 실현한다.

실례로 단정확도형옹근수파라메터가 100보다 큰가를 검사하는 함수는 다음과 같이 구성된다.

```
bind1st ( less<int>(), 100 )
```

함수적응자 bind1st의 구축자는 함수객체 greater<int>( )와 옹근수값 100을 기억한다. 그다음 함수적응자에 적용되면 실제파라메터 100과 999를 가진 함수객체 greater<int>()를 불러 내는 메쏘드 operator(999)가 호출된다. 그 처리를 그림 24-4에서 보여 준다.

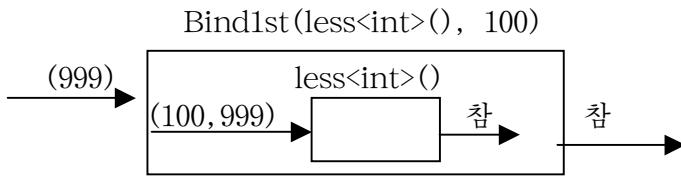



그림 24-4. 파라메터가 100보다 크면 참을 넘겨 주는 함수적응자

24.11.5 함수적응자 bind1st와 bind2nd의 리용

다음의 코드는 피보나치수렬에서 100보다 큰 첫번째 수를 찾는다.

```
int main ( )
{
  int number[15];                                              // std::vector
  const int BEGIN = 0;                                         // 첫번째 첨수
  const int SIZE = sizeof(numbers)/sizeof(int);                // 요소가 없다
  const int END =SIZE;                                         // 마지막첨수
  std::generate( &numbers[BEGIN], &numbers[END], fibonacci<int> ( ) );
  int *p = NULL;
//x>100
  p=std::fin_if(&numbers[BEGIN], &numbers[END],
                    std::bind2nd(std::greater<int>( ), 100 ) );
  std::cout << "first number in sequence greater than 100 is" <<*p<< "\n";
  return 0;
}
```

주의: 100보다 큰 수가 없으면 검사하지 않는다.

해당선언들과 함께 콤파일, 실행된 결과는 다음과 같다.

```
first number in sequence greater than 100 is 144
```

bind1st를 사용하여 100보다 큰 첫수를 찾는 코드는

```
p = std::find_if( &numbers[BEGIN], &number[END],
std::bind1st( std::less<int>( ),100 ) );
```

와 같이 씌여 질수 있다.

24.12 STL서고

표준본보기서고(STL)는 많은 범용알고리듬들을 제공한다. 부록 5는 STL의 일부 알고리듬들과 함수객체들 그리고 함수적응자들을 렬거한다.

24.13 자체평가

- 범용알고리듬 sort를 사용하여 클라스 Account의 구체례를 정렬할 때 이 본보기함수를 쓰는 사용자는 왜 콤파일시 오유통보문을 받게 되는가?
- 범용알고리듬사용에서의 우점은 무엇이며 부족점은 무엇인가?
- 함수객체란 무엇이며 어떻게 리용되는가?
- 함수적응자란 무엇이며 어떻게 리용되는가?`

24.14 련 습

지적자의미에 기초한 다음의 범용알고리듬들을 작성하시오.

- void count (I_It first, I_It last, const T& val, size& count)
  이것은 val에 정합되는 매 요소에 대한 증가계수를 진행한다.
- void count_if(I_ It first, I_It last, predicate f, size& count)
  이것은 bool f (I_It element)로 서술되는 함수 f에 의하여 정의되는 Predicate에 정합되는 매 요소에 대한 증가계수를 진행한다.
  이 범용알고리듬에서 형 I_It는 입력반복자이며 집합의 요소를 지적하는 지적자이다.

다음의 함수적응자를 작성하시오.

- as_string
  이것은 옹근수형을 넘겨 주는 함수객체를 int형결과의 string형표현을 넘겨 주는 함수객체로 변환한다.

25 STL 용기

이 장에서는 STL(Standard Template Library : 표준본보기서고)안에 정의되여 있는 용기(container)들에 대해서 서술한다. 일부 경우 용기의 일반성으로 하여 자원리용에서 결함이 있기는 하지만 표준용기들을 사용하면 프로그람의 구조를 간단하게 할수 있다.

25.1 STL용기에 대한 소개

STL서고는 여러가지 각이한 형태의 용기들을 제공한다. 해당한 용기를 정확히 사용하려면 그의 접근효과, 요소들의 삽입과 삭제능력, 기억기리용 등의 많은 인자들을 참고해야 한다.

용 기	특 징
vector	용기안의 요소들을 임의로 호출하고 용기끝에 새로운 요소들을 삽입한다.
list	용기안의 어디에나 요소들을 삽입하고 삭제한다.
deque	용기앞뒤에 요소들을 삽입하고 삭제한다.
set	요소들을 순서대로 놓고 용기에 대한 포함, 삽입, 삭제를 검사한다.
multiset	2중값을 가지는 모임.
map	열쇠(key)를 사용하여 값에 접근한다. 용기에 열쇠와 자료의 쌍들을 삽입하고 삭제한다.
multimap	2중열쇠를 가지는 map.
Stack	앞에서만 요소들을 삽입하고 삭제한다.
queue	앞에서는 항목을 삭제하고 뒤에는 항목을 삽입한다.
priority queue	용기에서 (최대/최소)값에 접근하고 삭제한다.

25.2 vector용기

vector용기클라스는 반복자(iterator)나 첨자(subscript)를 사용하여 요소들에 효과적으로 접근한다. 용기는 자기의 끝에 요소들을 추가하는것으로 확장된다. 사용자가 정하는 증가크기에 의해서 용기의 기억기가 확장된다. 일부 리용되지 않은 기억공간은 다음번 삽입에서 리용된다. 20.2에 서술된 클라스 Vector의 실현부는 vector용기의 특징들을 소규모적으로 실현한다.

vector용기로 실현되는 주요메쏘드들은 다음과 같다.

메쏘드	책 임
vector <T>	여기서 T는 집합체형이다. 구축자의 변종들: () 빈 용기 (size) 요소수 (size,value) size : 요소수, value : 초기내용
at (i)	i번째 요소를 돌려 준다. 만일 요소가 없다면 례외 out_of_range가 내보내진다.
back ( )	vector의 마지막요소를 돌려 준다.
begin ( )	vector의 첫번째 요소를 지적하는 **RAI**의 구체례를 돌려 준다.
empty ( )	만일 vector가 비여 있다면 참(true)을 돌려 준다.
end ( )	vector의 끝을 지나 첫번째 요소를 지적하는 **RAI**의 구체례를 돌려 준다.
erase(RAI) erase(RAI, RAI)	반복자로 지적된 vector에서 요소를 삭제하거나 반복자로 지정된 범위를 삭제한다. 삭제후 모든 반복자와 참조는 무효로 된다.
front ( )	vector의 앞요소를 돌려 준다.
insert (RAI, T)	insert(p, item)은 요소가 반복자 p로 지적되기전에 item을 삽입한다.
max_size ( )	사용되는 최대크기를 돌려 준다(이것은 일반적으로 기억된 요소들에 필요되는 기억기보다 더 많을수 있다).
operator =	vector의 복사를 실현한다.
operator [ ]	선택된 요소(이 요소는 표준첨자를 가진 연산자를 사용한다)의 참조를 돌려 준다.
pop_back ( )	vector의 끝요소를 삭제한다.
push_back (T)	vector의 끝에 새 요소를 추가한다.
rbegin ( )	vector의 시작에 역반복자(reverse iterator)를 돌려 준다.
rend ( )	vector의 끝에 역반복자를 돌려 준다.
reserve(size_type)	vector에 공간이 추가될 때 예약된 기억기크기를 설정한다.
size ( )	vector안의 요소수를 돌려 준다.

주의: vector용기는 머리부파일 <vector>에 정의되여 있다.
RAI는 임의접근반복자(Random Access Iterator)이다.

vector안에 매몰된 주요클라스들은 다음과 같다.

클라스	다음과 같은것을 설명하는데 리용된다
vector <T>::iterator	vector<T>구체례에 대한 읽기, 쓰기접근에 리용되는 **RAI**
vector<T>:: const_iterator	vector<T>구체례에 대한 읽기접근에만 리용되는 고정**RAI** **주의**: 고정용기인 경우에는 반드시 const _iterator 가 사용되여야 한다.
vector<T>::reverse_iterator	역**RAI**
vector<T>::const_reverse_iterator	읽기접근에만 리용되는 역**RAI**

vector용기안에 매몰된 주요클라스들은 다음과 같다.

클라스	설 명
vector<T>::value_type	vector안에 기억된 항목의 형
vector<T>::reference	용기안에 기억된 항목에 대한 참조
vector<T>::const_reference	용기안에 기억된 항목에 대한 고정참조
vector<T>::size_type	크기와 배렬첨수를 표현하는데 리용되는 형

25.2.1 vector용기를 사용한 삽입정렬

삽입정렬(insertion sort)에서는 항목들이 정렬된 항목집합안의 정확한 위치에 삽입된다. 일반적으로 이 알고리듬은 삽입연산의 복잡성으로 하여 자료정렬에 사용되지는 않는다. 삽입정렬을 리용하여 20, 10, 17, 18의 수들을 정렬하는 실례는 다음과 같다.

삽입되는 항목	삽 입 전	삽 입 후
20		20
10	20	10 20
17	10 20	10 17 20
18	10 17 20	10 17 18 20

vector용기의 구체례를 리용하는 이 자료정렬알고리듬의 실현을 아래에 보여 준다. 메쏘드 insert는 자료항목을 배렬안의 정확한 위치에 삽입한다. 정렬알고리듬자료는 문자렬흐름 cin의 구체례로부터 주어 진다.

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
typedef std::vector <int> Numbers;

int main( )
{
    char data[ ] = "20 10 17 18 15 11";
    std::istrstream cin ( data , strlen( data ) );              // 문자렬흐름

    Numbers  sorted( 0 );                                      // 요소 0들로 된 벡토르
    int   num;

    while ( cin >> num, !cin.eof( ) )
    {
        Numbers::size_type i = 0;                              // 초기위치
        while ( i < sorted.size( ) )                           // 배렬을 정렬
        {
            if ( sorted[ i ] > num )  break;                   // 삽입위치
            i ++;
        }
        sorted.insert ( &sorted[ i ], num );                    // 순서대로 삽입
    }

    std::cout << sorted[ i ] << " ";

    for ( unsigned int i=0; i<sorted.size( ); i++)              // 배렬인쇄
    {
        std::cout << sorted[ i ] << " ";                        // 배렬접근
     }
    std::cout << "\n";

    return 0;
}
```

주의: 용기안에 새 요소를 삽입하기 위해 메쏘드 insert를 사용한다. 이것은 새롭게 삽입되는 요소의 기억기를 만들기 위해 이미 있던 집합체구역안의 요소들을 이동해야 하므로 효률적이지 못하다.
클라스 vector는 이름공간 std안에 있다.
Numbers::size_type는 메쏘드size( )에 의해서 돌려 지는 값을 표시하는 형이다.

실행결과는 다음과 같다.

```
Numbers sorted are : 10 11 15 17 18 20
```

25.3 용기와 반복자

용기클라스들은 배렬에 대한 내장C++용기를 사용하는 방법과 류사한 방법으로 리용된다. 그러나 용기클라스들은 프로그람구조를 간단하게 하는 추가기능들을 제공한다. 용기클라스에서 정의된 클라스들은 지적자의미를 리용한 용기안의 요소들에 접근하기 위해서 리용되는 반복자들의 선언을 허용한다. 반복자들의 사용을 쉽게 하기

위해 용기클라스들은 다음의 표준연산자들을 재정의한 여러개의 반복자클라스들을 포함한다.

*　　지적된 값을 넘겨 주는 비참조반복자
+　　앞방향으로 n단위만큼 지적자를 전진
++　　앞방향으로 하나만큼 지적자를 전진
－　　반대방향으로 n단위만큼 지적자를 전진
－－　　반대방향으로 하나만큼 지적자를 전진

25.3.1 용기클라스에서 반복자

용기클라스들에서 반복자들은 사용에 해당한 계층으로 분류된다. 계층안의 어떤 위치에서 반복자들은 보다 아래계층의 반복자들이 허용되는 모든 연산자들을 지원한다. 반복자계층을 다음의 표에서 보여 준다.

반복자	속　성	
임의접근	임의의 위치에서 접근	가장 높다
쌍방향	앞 혹은 반대방향에서의 이동가능	
앞방향	앞방향으로만 이동가능	
입력/출력	흐름에 삽입자료를 추출하기 위해 리용됨	가장 낮다

25.3.2 지적자의미를 리용하는 삽입정렬

지적자의미를 리용하여 삽입정렬을 실현하기 위한 코드는 25.2.1에서 보여 준 코드와 비슷한 방법으로 작성된다. 다음의 코드에서 함수객체 print는 정렬된 배렬의 인쇄를 쉽게 하는데 리용된다.

```
template <class Type>
class print : std::unary_function<Type, void>          // 공통기초
{
public:
    void operator( )( const Type& x ) const
    {
        std::cout << x << " ";                         // 삽입연산자
    }
};
```

삽입정렬을 수행하는 실제코드는 다음과 같다.

```
#include < iostream >
#include < vector >
#include < algorithm >

typedef std::vector < int > Numbers;
```

```
int main( )
{
    char data[ ] = "20 10 17 18 15 11";
    std::istrstream cin ( data , strlen(data) );                // 문자렬흐름
    Numbers sorted(0);                                          // 요소 0들로 된 벡토르
    int   num;
    while ( cin >> num, !cin.eof( ) )
    {
        Numbers::iterator cur = sorted.begin( );                // 시작
        Numbers::iterator end = sorted.end( );                  // 끝
        while ( cur != end )                                    // 정렬하는 동안
        {
            if ( *cur > num ) break;                            // 삽입위치
            cur++;                                              // 다음위치
        }
        sorted.insert ( cur, num );                             // 순서대로 삽입
    }
    std::cout <<  "Numbers sorted are:  ";
    std::for_each( sorted.begin( ), sorted.end( ), print<int>( ) );
    std::cout << "\n";
    return 0;
}
```

주의: Numbers::iterator형을 사용하여 반복자를 선언한다.

실행결과는 다음과 같다.

```
Numbers sorted are : 10 11 15 17 18 20
```

25.3.3 고정반복자들을 리용

고정용기들을 리용한 경우에는 반드시 고정반복자들을 써서 그 용기의 값들을 반복해야 한다. 만일 일반반복자들을 쓰면 그에 의한 연산이 고정용기와 호환이 되지 않으므로 콤파일시에 오유통보문이 발생된다. 실례로 용기안의 모든 문자렬들을 인쇄하는 함수 print_strings는 다음과 같이 실현된다.

```
typedef std::vector<std::string> Strings;
void print_strings( const strings& col )
{
    Strings::const_iterator cur = col.begin( );
    Strings::const_iterator end = col.end( );
    while ( cur != end )                                        // 매개에 대하여
    {
```

```
        std::cout << *cur++ << " ";
    }
    std::cout << "\n";
}
```

25.3.4 용기들사이의 복사

copy알고리듬은 서로 다른 류형의 용기들사이에 자료를 전송하기 위해서 리용된다. 전송해야 할 자료량을 모를 때에는 뒤방향삽입자적응자가 새 자료값을 삽입하기 위해 용기의 push_back메쏘드를 호출하는 반복자를 발생하는데 리용된다. 실례로 벡토르(vector)용기에로 문자렬들의 배렬을 복사하기 위해서 다음의 코드가 사용된다.

```
//함수  print_strings (25.3.3을 보시오.)의 선언

int main( )
{
    Strings rainbow( 0 );
    std::string colours[ ] = {  "Violet", "Blue", "Green",
                                "Yellow", "Orange", "Red"  };
    const int BEGIN = 0;                                         // 첨자
    const int SIZE    = sizeof( colours ) / sizeof( std::string );   // 요소가 없다
    const int END     = SIZE;                                    // 첨자
    std::copy( &colours[ BEGIN ] , &colours[ END ] ,
std::back_inserter( rainbow ) );
    print_strings ( rainbow );
    return 0;
}
```

주의: 적응자 back_inserter(rainbow)는 용기를 확장하기 위해 용기의 push_back 메쏘드를 리용하는 반복자를 돌려 준다.

실행결과는 다음과 같다.

```
Violet Blue Green Yellow Orange Red
```

25.3.5 ostream_iterator적응자

ostream_iterator는 출력흐름에로 자료항목을 삽입하기 위해서 리용된다. 실례로 25.3.3에서 함수 print_strings는 다음과 같이 실현될수도 있었다.

```
typedef std::vector<std::string> Strings;
void  print_strings( ostream& ostr, const Strings& container )
{
    std::copy ( container.begin( ), container.end( ),
```

```
                  std::ostream_iterator<std::string>( ostr, " " ) );
    std::cout << "\n";
}
```

주의: 적응자 ostream_iterator는 용기안에 확보된 객체형으로서 파라메터화된다. 구축자의 두 파라메터는 다음과 같다.

ostream의 구체례

출력객체의 구체례들사이에 씌여 지는 분리기호

25.3.6 istream_iterator적응자

istream_iterator는 입력흐름으로부터 자료항목을 추출하기 위해서 리용된다. 실례로 다음의 연시프로그람은 문자렬흐름으로부터 용기 temperatures에로 옹근수값을 읽는다. 이 프로그람에서는 istream_iterator의 다음의 두 구체례가 리용된다.

istream_iterator의 구체례	설 명
ints(cin)	그 흐름에서 옹근수를 추출하는 흐름 cin의 반복자
eof	파일의 끝을 표현하는 반복자. 그 이름은 중요하지 않으며 중요한것은 객체가 파라메터를 가지지 않고 구성된다는것이다.

istream_iterator는 2개의 형으로서 파라메터화되여 있다.

- 그 흐름으로부터 추출하기 위한 값들의 형
- 2개의 지적자들사이의 차이를 표현하기 위한 형

```
int main( )
{
    typedef  std::vector<int>  Numbers;
    char data[ ] = "20 10 17 18 15 11";
    istrstream  cin ( data , strlen(data) );                    // 문자렬흐름

    Numbers  temperatures ( 0 );                                // temperatures집합

    std::istream_iterator< int , ptrdiff_t > ints( cin ), eof;
    std::copy(  ints,
                eof,
                std::back_inserter( temperatures ) );

    std::copy(  temperatures.begin( ), temperatures.end( ),
               std::ostream_iterator<int>( std::cout. " ") );
    return 0;
}
```

실행결과는 다음과 같다.

```
20 10 17 18 15 11
```

25.3.7 역반복자

메쏘드 rbegin과 rend는 역반복자들을 돌려 주는데 이것들은 본질적으로 일반반복자의 반대이다. 메쏘드 rbegin은 용기의 끝을 가리키는 반복자를 돌려 주며 rend는 용기의 첫 요소의 앞요소를 가리키는 반복자를 돌려 준다. 역반복자에서 연산자 ++는 앞의 요소에로 반복자를 전진시키며 연산자 --는 다음요소에로 반복자를 전진시킨다.

실례로 다음의 코드는 앞방향과 뒤방향순서로 vector의 내용을 출력한다.

```
#include <iostream>
#include <string>
#include <vector>

int main( )
{
    typedef std::vector< std::string > Strings;
    Strings  c( 0 );
    c.push_back( "Violet" );
    c.push_back( "Blue" );
    c.push_back( "Green" );
    c.push_back( "Yellow" );
    c.push_back( "Orange" );
    c.push_back( "Red" );

    std::for_each( c.begin( ) , c.end( ) , print< std::string >( ) );
    std::cout << "\n";

    std::for_each( c.rbegin( ) , c.rend( ) , print< std::string >( ) );
    std::cout << "\n";

    return 0;
}
```

실행결과는 다음과 같다.

```
Violet Blue Green Yellow Orange Red
Red Orange Yellow Green Blue Violet
```

25.4 vector용기의 구역검사추가법

구역검사는 새로운 클라스 vector를 파생하기 위해 계승을 리용하며 vector용기에 추가될수 있다. 새로운 클라스 vector안의 []연산자는 호출할 때 첨수검사를 제공하기 위해서 재정의된다.

```
#ifndef CLASS_VECTOR_SPEC
#define CLASS_VECTOR_SPEC

template <class Type>
class Vector  :  public std::vector<Type>
{
public:
    Vector( const size_type );
    const_reference operator [ ] ( const size_type ) const;
    reference operator [ ]( const size_type );
protected:
    void fail( char [ ] , const size_type ) const;
};

# endif
```

주의: 계승된 형들의 리용

reference	기억된 객체에 대한 참조를 위한것
const_reference	기억된 객체에 대한 고정참조를 위한것
size_type	배렬의 첨자를 서술하는 형을 위한것

이 클라스의 실현부는 다음과 같다.

```
#ifndef  CLASS_VECTOR_IMP
#define  CLASS_VECTOR_IMP
#include  <stdexcept>

template <class Type>
Vector<Type>::Vector( const size_type i ) : std::vector<Type>( i )
{
}
```

주의: 하나의 구축자만이 다중정의되였는데 만약 다른것들이 요구되였다면 그것들 역시 클라스 Vector안에서 다중정의되여야 한다.

[]연산자는 다중정의를 두번 진행하는데 앞의것은 고정객체(const object)를 위한것이고 다음의것은 변이객체(mutable object)를 위한것이다.

```
template <class Type>
const_reference Vector<Type>::operator[ ] ( const size_type i ) const
{
    if ( i < 0 || i >= size() )
          fail( "access", i );
    return std::vector<Type>::operator[ ] ( i );
}

template <class Type>
reference Vector<Type>::operator[ ] ( const size_type i )
```

```
{
    if ( i < 0 || i >= size( ) )
          fail( "access", i );
    return std::vector<Type>::operate[ ] ( i );
}
```

메쏘드 fail은 실패를 알리는 례외를 내보낸다.

```
template <class Type>
void Vector<Type>::fail( char mes[ ] , const size_type i ) const
{
    char storage[ 120 ];                                    // 통보문
    ostrstream text( storage,120 ) ;                        // 흐름으로서
    text << "Vector [Bounds 0.." << ( size( ) - 1 )<<
              " - " <<mes << " Subscript: " <<
              i << "]" << '\0' ;
    throw std::range_error( std::string(storage) );         // 례외
}
#endif
```

25.4.1 종합서술

다음의 코드는 첨자검사를 실현하는 클라스 Vector의 사용을 보여 준다.

```
// 새로운 Vector클라스의 포함
int main( )
{
     Vector<std::string> colours( 0 ) ;
     colours.push_back( "Red" );
     colours.push_back( "Blue" );
     colours.push_back( "Green" );
     colours.push_back( "Orange" );
     colours.push_back( "Yellow" );
     colours.push_back( "Violet" );
     try
     {
          std::cout << "Colours of rainbow : ";
          for ( Vector<std::string>::size_type i=0; i < colours.size( ); i++ )
          {
               std::cout <<colours[ i ] << " ";
          }
          std::cout << "\n";
          std::cout << "Colours of rainbow : ";
          for ( Vector<std::string>::size_type i=0; i < colours.size( ); i++)
          {
```

```
            std::cout << colours[ i ] << " ";
        }
        std::cout << "\n';
    }
    catch ( std::exception& err )
    {
        cerr << "\n" << "Failure: " << err.what( ) << "\n";
    }
    return 0;
}
```

실행결과는 다음과 같다.

```
Colours of rainbow : Red Blue Green Orange Yellow Violet
Colours of rainbow : Red Blue Green Orange Yellow Violet
Failure: Vector [Bound 0 .. 5 - access Subscript : 6]
```

25.5 list용기

list용기클라스는 반복자를 사용하는 요소들에 대한 순차접근을 제공한다. 용기는 목록의 어떤 부분에 추가적인 요소들을 덧붙이는것에 의해 효률적으로 확장될수 있다.

list용기로 실현되는 주요메쏘드들은 다음과 같다.

메쏘드	책　임
list<T>	여기서 T는 집합형이다. 구축자의 변종들: ( )　　빈 용기 (size)　　요소들의 수를 표시 (size, value)　size:요소수, value:초기내용
back( )	목록의 마지막요소를 돌려 준다.
begin( )	목록의 시작을 지정하는 쌍방향반복자를 돌려 준다.
empty( )	목록이 비여 있다면 참을 돌려 준다.
end ( )	목록의 끝을 지나서 첫번째 요소를 지정하는 쌍방향반복자를 돌려 준다.
erase(BDI)	반복자에 의해 지적된 목록으로부터 요소를 삭제한다.△
front( )	목록의 앞요소를 돌려 준다.
insert(BDI,T)	반복자앞의 목록안에 요소를 삽입한다. △
max_size( )	목록의 가능한 최대크기를 돌려 준다.
operator=	목록의 복사를 실현한다.

메쏘드	책　임
pop_back( )	목록의 뒤끝에서 요소를 삭제한다.
pop_front( )	목록의 앞끝에서 요소를 삭제한다.
push_back (T)	목록의 뒤끝에 새 요소를 추가한다.
push_front(T)	목록의 앞끝에 새 요소를 추가한다.
rbegin( )	목록의 시작위치에로 역반복자를 돌려 준다.
rend( )	목록의 시작위치에로 역반복자를 돌려 준다.
size( )	목록안에 있는 요소들의 수를 돌려 준다.

주의: list용기는 머리부파일 <list>안에 정의된다.
BDI는 쌍방향반복자이다.
△ : 모든 반복자들과 참조들은 삭제 혹은 삽입후에 무효로 된다.

list용기는 다음것을 제외하고는 vector용기와 매우 비슷하다.

- 첨자 있는 연산자 []는 사용할수 없다.
- push_front와 pop_front의 추가메쏘드들은 목록의 앞에 대하여 자료를 추가하고 삭제한다.

클라스 list안에 들어 있는 클라스들은 다음과 같다.

클라스	아래와 같은것을 선언하는데 리용된다
list<T>::iterator	list<T>의 구체례에 대한 읽기, 쓰기접근을 위해서 리용되는 쌍방향반복자
list<T>::const_iterator	list<T>의 구체례에 대한 읽기호출을 위해 리용되는 고정쌍방향반복자 **주의**: 고정용기인 경우에는 반드시 const_iterator가 리용되여야 한다
list<T>::reverse_iterator	역쌍방향반복자
list<T>::const_reverse_iterator	고정역쌍방향반복자

목록용기안에 들어 있는 클라스들은 다음과 같다.

클라스	설　명
List<T>::value_type	용기안에 기억된 항목의 형
List<T>::reference	용기안에 기억된 항목에 대한 참조
List<T>::const_reference	용기안에 기억된 항목에 대한 고정참조
List<T>::size_type	크기를 나타내는데 리용하는 형

25.5.1 지적자의미를 리용하는 삽입정렬

실례로 목록용기를 리용하는 삽입정렬알고리듬을 실현하고 그 결과를 인쇄하는 프로그람을 아래에 보여 준다.

```
#include <iostream>
#include <list>
#include <algorithm>

typedef  std::list <int> Numbers;

int main( )
{
    char data[ ] = " 20 10 17 18 15 11 ";
    istrstream cin ( data , strlen(data) );                    //문자렬흐름

    Numbers     sorted( 0 );                                   // 0 들로 된 요소벡토르
    int         num;

    while ( cin >> num, !cin.eof( ) )
    {
        Numbers::iterator cur = sorted.begin( );               // 시작
        Numbers::iterator end = sorted.end( );                 // 끝
        while ( cur != end )                                   // 정렬하는 동안
        {
            if ( *cur > num ) break;                           // 삽입위치
            cur++;                                             // 다음
        }
        sorted.insert ( cur, num );                            // 순서로 삽입
    }

    std::cout <<  "Numbers sorted are:  ";

    std::for_each( sorted.begin( ), sorted.end( ), print<int>( ) );
    std::cout << "\n";
    return 0;
}
```

주의: 이것은 형 Numbers 가 list 용기의 구체례라는것을 제외하고는 vector 용기를 사용한 이전의 삽입정렬코드와 같은 코드이다.

실행결과는 다음과 같다.

```
Numbers sorted are : 10 11 15 17 18 20
```

25.5.2 용기들사이의 복사

목록용기가 목록의 앞에서 자료의 삽입을 지원하므로 front_inserter적응자는 자료를 삽입하는데 리용될수 있다. 실례로 다음의 코드는 목록 rainbow의 앞에 무지개색들을 복사한다.

```
typedef std::list<std::string> Strings;
int main( )
{
    Strings rainbow( 0 );
    std::string colours[ ] = { "Violet", "Blue", "Green", "Yellow", "Orange", "Red" };
    const int BEGIN = 0;                                              // 색인
    const int SIZE    = sizeof( colours ) / sizeof( std::string );    // 요소가 없다
    const int END     = SIZE;                                         // 색인
    std::copy( &colours[ BEGIN ] , &colours[ END ] ,
                    std::front_inserter( rainbow ) );
    std::copy( rainbow.start( ) , rainbow.end( ) ,
                    std::ostream_iterator<std::string>( std::cout , " " ) );
    std::cout << “\n”;
    return 0;
}
```

실행결과는 다음과 같다.

```
Red Orange Yellow Green Blue Violet
```

25.6 deque용기

deque용기클라스는 vector와 list용기사이의 교차이다. 이 용기는 데쿠(deque : 량끝에서 자료접근이 가능한 자료대기렬)의 앞뒤에서 요소들에 대한 임의접근과 효과적인 삽입을 진행한다. deque용기로 실현되는 주요메쏘드들은 다음과 같다.

메쏘드	책　임
deque<T>	여기서 T는 집합형이다. 구축자의 변종들: ( )　　빈 용기 (size)　　요소들의 수를 표시 (size, value)　　size: 요소수, value: 초기내용
at (i)	i번째 요소를 돌려 준다. 요소가 존재하지 않는다면 례외 out_of_range가 발생된다.

메쏘드	책 임
back(i)	데쿠의 마지막요소를 돌려 준다.
begin( )	데쿠의 시작을 지적하는 임의접근반복자를 돌려 준다.
empty( )	만약 데쿠가 비여 있다면 참을 돌려 준다.
end( )	데쿠의 끝을 지나서 첫번째 요소를 지적하는 임의접근반복자를 돌려 준다.
erase(RAI)	반복자에 의하여 지적된 데쿠로부터 요소를 삭제한다. △
front( )	데쿠의 앞요소를 돌려 준다.
insert(RAI,T)	반복자전에 데쿠안에 요소를 삽입한다. △
max_size( )	데쿠의 가능한 최대크기를 돌려 준다.
operator =	데쿠의 복사를 실현한다.
operator[ ]	일반적인 첨수 달린 연산자를 사용할수 있도록 선택된 요소의 참조를 돌려 준다.
pop_back( )	데쿠의 뒤끝에서 요소를 삭제한다.
pop_front( )	데쿠의 앞끝에서 요소를 삭제한다.
push_back(T)	데쿠의 뒤끝에 새 요소를 추가한다.
push_front(T)	데쿠의 앞끝에 새 요소를 추가한다.
rbegin( )	데쿠의 시작위치에 역반복자를 돌려 준다.
rend( )	데쿠의 시작위치에 역반복자를 돌려 준다.
size( )	데쿠안의 요소수를 돌려 준다.

주의: deque용기는 머리부파일 <deque>안에 정의된다.
RAI는 임의접근반복자이다.
△는 모든 반복자들과 참조들은 삭제 혹은 삽입후에 무효로 된다는 뜻이다.

용기 deque안에 들어 있는 주요클라스들은 다음과 같다.

클라스	아래와 같은것을 선언하는데 리용된다
deque<T>::iterator	deque<T>의 구체례에 대한 읽기와 쓰기접근에 사용된 쌍방향반복자
deque<T>::const_iterator	deque<T>의 구체례에 대한 읽기접근에 사용된 고정적인 쌍방향반복자 **주의**: 고정용기인 경우에는 const_iterator가 리용되여야 한다.
deque<T>::reverse_iterator	역쌍방향반복자
deque<T>::const_reverse_iterator	고정역쌍방향반복자

deque용기안에 매몰된 주요클라스들은 다음과 같다

클라스	설 명
deque<T>::value_type	용기안에 기억된 항목의 형
deque<T>::reference	용기안에 기억된 항목의 참조
deque<T>::const_reference	용기안에 기억된 항목의 고정참조
deque<T>::size_type	크기를 나타내는데 리용하는 형

다음의 실례프로그람은 삽입정렬방법으로 수들을 정렬시키기 위해서 데쿠를 사용한다. 데쿠안의 수들은 첨자 달린 일반연산자를 사용하여 인쇄된다.

```
#include <string>
#include <deque>
#include <algorithm>
#include <iostream>
#include <strstream>
int main( )
{
    typedef  std::deque <int>  Numbers;
    char data[ ] = " 20 10 17 18 15 11 ";
    istrstream cin ( data , strlen(data) );                    // 문자렬흐름
    Numbers  sorted( 0 );                                      // 요소 0들로 된 벡토르
    int          num;
    while ( cin >> num, !cin.eof( ) )
    {
          Numbers::iterator cur = sorted.begin( );             // 시작
          Numbers::iterator end = sorted.end( );               // 끝
          while ( cur != end )                                 // 정렬하는 동안
          {
              if ( *cur > num )  break;                        // 삽입위치
              cur ++;                                          // 다음
          }
          sorted.insert ( cur, num );                          // 순서대로 삽입
    }
    std::cout <<  "Numbers sorted are:  ";
    for ( Numbers::size_type i = 0; i < sorted.size( ) ; i++ )
    {
          std::cout << sorted[ i ] << " ";
    }
    std::cout << "\n";
    return  0;
}
```

실행결과는 다음과 같다.

```
Numbers sorted are : 10 11 15 17 18 20
```

25.7 stack용기

stack용기클라스는 선입후출(first in last out)접근기구를 리용하여 자료항목들을 기억하고 검색한다.

stack용기로 실현되는 주요메쏘드들은 다음과 같다.

메쏘드	책　임
stack<T,Con> stack<T>	여기서 T는 집합형이다. Con은 자료항목을 기억하는데 리용되는 용기이다. 구축자의 변종: ( )　빈 용기
empty( )	탄창이 비여 있다면 참을 돌려 준다.
operator=	탄창의 복사를 실현한다.
pop( )	탄창으로부터 꼭대기항목을 삭제한다.
push(T)	탄창에 새 요소를 추가한다.
size( )	탄창안의 요소들의 수를 돌려 준다.
top( )	탄창의 꼭대기항목을 돌려 주지만 그것을 삭제하지 않는다.

stack용기안에 들어 있는 클라스는 다음과 같다.

클라스	설　명
stack<T,con>::value_type	용기안에 기억된 항목의 형

25.8 queue용기

queue용기클라스는 선입선출(first in first out)접근기구를 리용하여 자료항목들을 기억하고 검색한다.

queue용기로 실현되는 주요메쏘드들은 다음과 같다.

메쏘드	책　임
queue<T,Con>	여기서 T는 집합형이다. Con은 자료항목들을 기억하는데 리용되는 용기이다. 구축자의 변종 ():　빈 용기

메쏘드	책　임
back( )	대기렬의 끝에 있는 요소를 돌려 준다.
empty( )	만일 대기렬이 비였다면 참을 돌려 준다.
front( )	대기렬의 앞에 있는 요소를 돌려 주지만 그것을 삭제하지는 않는다.
pop( )	대기렬의 앞에 있는 요소를 삭제한다.
push(T)	대기렬의 끝에 새로운 요소를 추가한다.
size( )	대기렬안의 요소들의 수를 돌려 준다.

queue용기안에 들어 있는 클라스는 다음과 같다.

클라스	설　명
queue<T,Con>:: value_type	용기안에 기억된 항목의 형

25.9 priority_queue용기

priority_queue용기클라스는 집합안의 가장 작은 또는 가장 큰 항목에 대한 고속검색을 진행한다.

priority_queue용기로 실현되는 주요메쏘드들은 다음과 같다.

메쏘드	책　임
priority_queue <T,Con,Cmp>	여기서 T는 집합형이다. Con은 자료항목을 기억하는데 리용되는 용기이다. Cmp는 우선권대기렬로 항목들을 정돈하기 위한 함수객체이다. 구축자변종: ( ) 빈 용기
empty( )	만일 우선권대기렬이 비였다면 참을 돌려 준다.
top( )	우선권대기렬의 앞에 있는 요소를 돌려 주지만 그것을 삭제하지는 못한다.
pop( )	우선권대기렬의 앞에 있는 요소를 삭제한다.
push(T)	우선권대기렬의 끝에 새로운 요소를 추가한다.
size( )	우선권대기렬의 요소들의 수를 돌려 준다.

priority_queue용기안에 매몰된 기본형은 다음과 같다.

클라스	설　명
priority_queue<T,Con,Cmp>::value_type	용기안에 기억된 항목의 형

25.9.1 priority_queue를 사용한 정렬프로그람

다음의 증시프로그람은 우선권대기렬(priority queue)집합안에 수들을 읽어 들인 다음 그 모임에서 가장 작은 수를 골라 목록의 끝에 놓는다.

priority_queue의 실례에서 쓰이는 파라메터들은 다음과 같다.

- 기억된 항목의 형
- 자료항목들을 기억하는데 리용되는 용기
- 우선권대기렬로 자료항목들을 정돈하기 위해 리용되는 함수객체. 순서는 메쏘드 top를 사용하여 삭제되는 가장 낮은 대조항목을 결정한다.

```
#include <iostream>
#include <string>
#include <queue>                    // 우선권대기렬과 대기렬
#include <vector>                   // 우선권대기렬로 사용되는 용기
#include <list>
#include <algorithm>
#include <strstream>

tempate <class Type>
class print:std::unary_function <Type , void>            // 공통기초
{
public:
   void operator( ) ( const Type& x ) const
    {
          std::cout << x << " ";                          // 추출연산자
    }
};
```

```
int main( )
{
    typedef  std::priority_queue < int , std::vector<int> ,
    std::greater<int> > Pq;
    typedef  std::list <int>  Numbers:
    Pq  numbers;                                  // 수들로 된 우선권대기렬
    Numbers  sorted;                              // 0요소들의 목록
    char data [ ] = " 20 10 17 18 15 11 ";
    istrstream  cin ( data , strlen ( data ) );   // 문자렬흐름

    int   num;

    while ( cin >> num , !cin.eof ( ) )           // P대기렬안에 수를 넣기
    {
          numbers.push ( num );
    }

    while ( numbers.size ( ) > 0 )
```

```
    {
        sorted.push_back ( numbers.top ( ) );       // 가장 작은것을 추가(목록)
        numbers.pop ( ) ;                           // 가장 작은것을 삭제(우선권대기렬)
    }
    std::cout << "Numbers sorted are: ";
    std::for_each( sorted.begin( ), sorted.end( ), print<int>( ) );
    std::cout << "\n";
    return 0;
}
```

이것을 콤파일하고 실행하면 다음의 결과를 얻는다.

```
Numbers sorted are : 10 11 15 17 18 20
```

정렬순서는 priority_queue선언을 아래와 같이 변경함으로써 반대로 할수 있다.

```
typedef std::priority_queue <int, std::vector<int>, std::less<int>> Pq;
```

25.10 map용기와 multimap용기

map와 multimap용기클라스들은 열쇠자료를 사용하여 정보의 기억과 검색을 실현한다. 도표쌍(열쇠, 자료)은 열쇠를 사용하여 기억되고 검색된다.

map와 multimap용기들로 실현되는 주요메쏘드들은 다음과 같다.

메쏘드	책　임
map<T1, T2, F0>	여기서 T1은 열쇠의 형이다. T2는 열쇠와 련관되는 항목의 형이다. F0은 집합안의 항목들을 정돈하는데 리용되는 함수객체이다. 구축자변종: ( )　빈 도표
begin( )	도표(map)의 시작을 가리키는 **RAI**를 돌려 준다.
empty( )	만약 도표가 비여 있다면 참을 돌려 준다.
end( )	도표끝요소의 다음요소를 지적하는 쌍방향반복자를 돌려 준다.
erase(T1)	도표로부터 열쇠를 삭제한다.
find(T1)	(열쇠, 자료)쌍에로 쌍방향반복자를 돌려 준다.
insert(VT)	도표안에 요소를 삽입한다.

메쏘드	책 임
max_size( )	도표의 가능한 최대크기를 돌려 준다.
Operator=	도표의 복사를 실현한다.
Operator[]	일반적인 첨자 달린 연산자를 쓸수 있도록 선택된 열쇠의 참조를 돌려 준다.
Rbegin( )	역방향으로 도표를 검색하는데 사용되는 쌍방향반복자를 돌려 준다.
rend( )	역방향으로 도표를 검색하는데 사용되는 쌍방향반복자를 돌려 준다.
size( )	도표안의 요소수를 돌려 준다.

주의: vector map는 머리부파일 <map>에 정의되여 있다.
VT는 클라스 map<T!,T2,F0>::value_type이며 이 클라스의 구체례는 (열쇠, 자료)쌍을 기억한다. 실례로
map<T1,T2,F0>::value_type (T1의 구체례,T2의 구체례)

클라스	아래와 같은것을 선언하는데 리용된다
map <T1,T2,F0> :: iterator	map <T1, T2, F0>의 구체례에 대한 읽기, 쓰기접근에 리용되는 쌍방향반복자
map<T1,T2,F0>::const_iterator	map <T1, T2, F0> 구체례의 읽기접근에만 리용되는 쌍방향반복자 . **주의**: 고정용기인 경우에는 const_iterator를 리용하여야 한다.

map용기안에 들어 있는 클라스는 다음과 같다.

형	설 명
map<T1,T2,F0>::value_type	도표안에 기억된 값들의 형 쌍 <**const** key, Data>들의 개별적인 성원들은 first와 second이름을 사용하여 선택된다.
map<T1,T2,F0>::key_type	도표안의 열쇠의 형

25.10.1 map용기의 리용

서로 다른 색을 가진 승용차들의 번호를 기록하는 승용차대리점은 재고(stock)를 가지고 있다. 도표 stock는 다음의 형정의명령문을 사용하는 string형첨자마당과 int형자료마당을 가지고 만들어 진다.

```
typedef map< std::string , int , std::less<string> > Stock;
```

주의: map의 파라메터들은 다음과 같다.

- 기억된 자료를 호출하기 위해 리용된 첨자
- 첨자를 가지고 기억된 자료
- 첨자구체레들사이의 <의 정의. 이것은 map안에 가지고 있는 열쇠들의 대조결과를 결정한다.

우의 실례에서 문자렬들사이 <의 정의는 함수객체 less<std::string>로 제공된다. 24.9에서는 다른 함수객체들을 서술한다.

다음과 같이 첨자 달린 일반연산들을 리용하여 map용기안에 자료를 입력할수 있다.

```
#include <iostream>
#include <string>
#include <algorithm>
#include <map>

typedef  map< std::string , int , std::less<string> > Stock;

int main ( )
{
    Stock cars;

    cars[ "red"]          = 2;                       // 초기재고
    cars[ "blue"]         = 5;
    cars[ "silver"]       = 4;
    cars[ "green"]        = 4;

    cars[ "red"] = cars[ "red" ] + 2 ;               // 2대의 붉은색승용차를 추가

    std::for_each( cars.begin( ) , cars.end( ) , print_car );
    std::cout << "\n";
    return 0;
}
```

Stock집합안의 매 요소에 적용된 함수 print_car는 다음과 같이 실현된다.

```
void  print _car ( Stock::value_type& entry )
{
    std::cout << "There are " << std::setw( 3 ) << entry.second
                   << " " << entry.first << " cars" << "\n";
}
```

주의: 집합안의 매 항목은 (열쇠, 자료)쌍으로서 기억된다. 쌍의 구체레는 함수객체 stock::value_type를 사용하여 만들어 진다. 쌍의 개별적인 성원들은 이름 first와 second를 사용하여 선택된다.

실행결과는 다음과 같다.

```
There are       5 blue cars
There are       4 green cars
There are       4 red cars
There are       4 silver cars
```

코드는 다음과 같이 (열쇠, 자료)쌍의 명시적인 참조를 리용하여 작성된다.

```
#include < iostream >
#include < string >
#include < algorithm >
#include < map >

int main ( )
{
    Stock  cars;
    cars.insert ( Stock::value_type ( "red" , 2 ) );              // 초기재고
    cars.insert ( Stock::value_type ( "blue" , 5 ) );
    cars.insert ( Stock::value_type ( "silver" , 4 ) );
    cars.insert ( Stock::value_type ( "green" , 4 ) );
    cars.erase( "red" );                                          // 갱신
    cars.insert( Stock::value_type( "red",4 ) );

    Stock::iterator p = cars.begin( );

    while ( p != cars.end( ) )
    {
        std::cout << "There are " << std::setw( 3 ) << (*p).second
                  << " " << (*p).first << " cars" << "\n";
        p ++;
    }
    return 0;
}
```

실행결과는 다음과 같다.

```
There are       5 blue cars
There are       4 green cars
There are       4 red cars
There are       4 silver cars
```

25.10.2 항목이 도표안에 있는가를 검사하기

다음의 함수는 어떤 승용차색이 25.10.1에서 리용하였던 도표 stock의 구체레안에 기억되여 있는가를 검사하는 메쏘드 find를 리용한다. 만일 그 색이 도표(map)안에 포함되여 있다면 그때 재고(stock)안에 있는 그 지정된 색을 가진 승용차들의 번호가 인쇄되며 그렇지 않으면 통보문 《그런 항목은 재고에 없다.》가 인쇄된다.

```
void how_many ( Stock& cars, std::string colour )
{
    if ( cars.find( colour ) != cars.end( ) )
    {
          std::cout << "There are " << std::setw( 3 ) << cars[ colour ]
                               << " " << colour << " cars" << " \n ";
    } else {
          std::cout << "There are no " << colour << " cars in stock " << " \n ";
    }
}
```

아래의 증시프로그람은 승용차들에 대한 도표(map)를 서술하며 그때 도표에 질문하는 함수 how_many를 리용한다. 그리고 메쏘드 erase는 도표에서 항목들을 삭제하기 위해 리용된다.

```
#include < iostream >
#include < string >
#include < algorithm >
#include < map >

typedef  std :: map< std::string , int , std::less<std :: string> > Stock;

// 함수 how_many, print_car

int main ( )
{
    cars [ " red " ]       = 2;                   cars [ " blue " ]   = 5;
    cars [ " silver " ]  = 4;                     cars [ " green " ] = 4;

    std::cout << " List of cars " << " \n ";
    std::for_each ( cars.begin( ) , cars.end ( ) , print_car );
    std::cout << " \n " << " Interrogate and manipulate stock " << " \n ";
    how_many ( cars, " red " );                     // 붉은색차는 얼마나 있는가
    how_many ( cars, " pink " );                    // 분홍색차는 얼마나 있는가
    cars.erase ( " red " );                         // 도표에서 붉은색차들을 삭제
    how_many ( cars, " red " );                     // 붉은색차가 얼마나 있는가
    std::cout << " \n " << " List of cars " << " \n ";
    std::for_each ( cars.begin( ) , cars.end ( ) , print_car );
    return 0;
}
```

실행결과는 다음과 같다.

```
List of cars
There are         5 blue cars
There are         4 green cars
```

```
There are        2 red cars
There are        4 silver cars

Interrogate and manipulate stock
There are        2 red cars
There are no pink cars in stock
There are no red cars in stock

List of cars
There are        5 blue cars
There are        4 green cars
There are        4 silver cars
```

25.10.3 다중도표의 사용

다중도표(multimap)는 같은 열쇠를 다중복사할수 있다는것을 제외하고는 도표(map)와 같다. 실례로 다음의 프로그람은 풀색열쇠가 2번 발생하도록 되여 있는 승용차들의 색을 가지는 도표를 서술한다.

메쏘드 find가 풀색열쇠를 알아 내기 위해서 리용되며 그때 풀색인 모든 열쇠들을 인쇄하는 순환이 실행된다. 도표에서 풀색이 삭제된후 그의 열쇠도 다같이 삭제된다.

```
typedef std::multimap< std::string , int , std::less<std::string> > Stock;

int main ( )
{
    Stock cars;

    cars.insert ( Stock::value_type ( " red " , 2 ) );          // 초기화
    cars.insert ( Stock::value_type ( " blue " , 5 ) );
    cars.insert ( Stock::value_type ( " green " , 4 ) );
    cars.insert ( Stock::value_type ( " green " , 8 ) );        // 열쇠를 복사
    cars.insert ( Stock::value_type ( " silver " , 4 ) );

    std::cout << " List all keys in multimap " << " \n ";

    Stock::iterator p = cars.begin( );
    while ( p != cars.end( ) )
    {
        std::cout << "There are " << std::setw( 3 ) << (*p).second
                  << " " << (*p).first << " cars" << "\n";
        p++;
    }
    std::cout << " \n " << " Print multiple keys for green cars " << " \n ";

    std::strings key = " green ";
    p = cars.find( key );
    while ( p != cars.end( ) && ( *p ).first == key )
```

```
    {
          std::cout << "There are " << std::setw( 3 ) << (*p).second
                        << " " << (*p).first << " cars" << "\n";
          p++;
    }
    cars.erase ( " green ");
    std::cout << "\n" << " Erase green key " << "\n" << " List all keys in multimap "
<< "\n";
    p = cars.begin ( );
    while ( p != cars.end( ) )
    {
         std::cout << "There are " << std::setw( 3 ) << (*p).second  << " "
                                     << (*p).first << " cars" << "\n";
          p++;
    }
    return 0;
}
```

실행결과는 다음과 같다.

```
List all keys in multimap
There are          5 blue cars
There are          4 green cars
There are          8 green cars
There are          2 red cars
There are          4 silver cars
Print multiple keys for green cars
There are          4 green cars
There are   8 green cars
Erase green key
List all keys in multimap
There are          5 blue cars
There are          2 red cars
There are          4 silver cars
```

25.11 set용기와 multiset용기

set와 multiset용기클라스들은 단일자료항목들의 기억과 검색을 실현한다. 어떤 모임(set)안의 항목들은 그 모임이 구성될 때 결정되는 함수객체를 리용하여 정돈된다.

set와 multiset용기들에 의하여 실현되는 주요메쏘드들은 다음과 같다.

메쏘드	책　임
set<T,F0>	여기서 T는 집합형이다. F0은 집합안의 항목들을 정돈하기 위해 사용되는 함수객체이다 구축자변종: 　　( )　빈 모임
begin( )	모임의 시작을 지적하는 쌍방향반복자를 돌려 준다.
empty( )	모임이 비여 있다면 참을 돌려 준다.
end( )	모임의 끝을 지나서 첫 요소를 지적하는 쌍방향반복자를 돌려 준다.
erase(T)	모임으로부터 요소(들)를 삭제한다.
find(T)	요소에 쌍방향반복자를 돌려 준다.
insert(T)	모임안에 요소를 삽입한다.
max_size( )	모임의 가능한 최대크기를 돌려 준다.
operator=	모임의 복사를 실현한다.
rbegin( )	반대방향에서 모임을 횡단하기 위해 리용된 쌍방향반복자를 돌려 준다.
rend( )	반대방향에서 모임을 횡단하기 위해 리용된 쌍방향반복자를 돌려 준다.
size( )	모임안의 요소들의 수를 돌려 준다.

주의: 모임은 머리부파일 <set>안에 정의된다.

클라스	아래와 같은것을 선언하는데 리용된다
set<T,F0>::iterator	set<T,F0>의 구체례에 대한 읽기, 쓰기접근을 위해 사용되는 쌍방향반복자
set<T,F0>::const_iterator	set<T,F0>의 구체례에 대한 읽기접근을 위해 리용되는 고정쌍방향반복자 **주의**: 고정용기인 경우에는 const_iterator를 리용하여야 한다.

모임용기에 들어 있는 주요류형은 다음과 같다.

형	설　명
set<T,F0>::type	모임안에 기억된 값들의 형

25.11.1 set용기의 리용

실례로 문자렬을 보유하는 모임(set)은 다음과 같이 정의된다.

```
typedef std::set< std::string , std::less<std::string> > Strings;
```

주의: set의 파라메터들은 다음과 같다.

- set안에 기억된 항목의 형
- set의 구체례들사이에 있는 <의 정의

우의 실례에서 문자렬들사이의 <의 정의는 함수객체 less<std::string>에 의하여 제공된다.

다음의 실례프로그람은 서로 다른 색들을 가지는 모임을 서술하며 그때 무지개색이 아닌 색을 삭제한다.

```
typedef  std::set< std::string , std::less<std::string> > Strings;

int main( )
{
    Strings colours;
    colours.insert ( " Violet " );   colours.insert ( " Blue " );
    colours.insert ( " Green " );    colours.insert ( " Yellow " );
    colours.insert ( " Orange " );  colours.insert ( " Red " );
    colours.insert ( " Olive " );

    std::copy ( colours.begin( ) , colours.end( ),
                        std::ostream_iterator<std::string> ( std::cout , " " ) );
    std::cout << " \n ";
    colours.erase ( std::string( " Olive " ) );
    std::copy ( colours.begin( ) , colours.end( ),
                        std::ostream_iterator<std::string> ( std::cout , " " ) );
    std::cout << " \n ";
    return 0;
}
```

실행결과는 다음과 같다.

```
Blue Green Olive Orange Red Violet Yellow
Blue Green Orange Red Violet Yellow
```

25.11.2 **multiset**용기의 리용

다중모임(multiset)은 한 항목을 여러번 복사할수 있다는것을 제외하고 모임(set)과 같다. 실례로 아래의 프로그람에서는 무지개색갈모임이 들어 있는데 이 모임에서는 붉은색이 두번 나타난다. 이 붉은색이 모임에서 제거되며 결과적으로 이 색의 두번발생이 제거된다.

```
typedef std::multiset < std::string , std::less< std::strings > > Strings;

int main( )
{
    Strings colours;
    colours.insert ( " Violet " );   colours.insert ( " Blue " );
    colours.insert ( " Green " );    colours.insert ( " Yellow " );
    colours.insert ( " Orange " );  colours.insert ( " Red " );
    colours.insert ( " Olive " );

    std::copy ( colours.begin( ) , colours.end( ) ,
                    std::ostream_iterator<std::string> ( std::cout , " " ) );
    std::cout << " \n ";
    colours.erase ( std::string( " Red " ) );             //모든 붉은색요소를 없앤다
    std::copy ( colours.begin( ) , colours.end( ),
                    std::ostream_iterator<std::string> ( std::cout , " " ) );
    std::cout << " \n ";
    return 0;
}
```

실행결과는 다음과 같다.

```
Blue Green Orange Red Red Violet Yellow
Blue Green Orange Violet Yellow
```

25.12 자체평가

- 프로그람안에서 용기클라스를 사용하는것이 가지는 우점은 무엇인가?
- 어떤 환경에서 벡토르용기대신에 목록용기를 쓰는것이 더 좋은가. 그 반대경우는 언제 성립하는가?

- 만약 용기안에 객체의 구체례가 기억된다면 무슨 연산들이 지원되여야 하며 왜 그런가?
- 사용자는 왜 용기안의 지적자들을 기억하는데 심중해야 하는가?

25.13 련 습

다음의것들을 만드시오.

- 구좌
  거래를 검열추적기구의 부분으로서 기억하는 은행구좌를 나타내는 클라스. 그 은행클라스들은 은행에 결합된 일반성원함수들외에 다음의 성원함수들을 추가하여 실현한다.

메쏘드	책 임
clear_audit_trail	audit trail을 지운다.
return_audit_trail	audit trail요소들의 집합을 돌려 준다.

- 은행
  개별적인 구좌들이 적당한 용기안에 기억되여 있는 은행을 나타내는 클라스. 은행클라스는 새로운 구좌들의 추가와 일반은행업무처리를 지원한다.
- 맞춤법검사
  적당한 용기들을 리용하는 간단한 맞춤법검사기를 작성하시오. 맞춤법검사를 위한 간단한 방법은 정확한 단어사전을 가지는 용기를 사용하는것이다. 만일 단어사전에 있다면 정확한 글자로 간주된다. 사용자가 요구하는 단어가 정확한데 사전에 그것이 없다면 이때 단어는 사용자의 사전에 추가된다.

26 C++유산콤파일러의 리용

이 장에서는 선행한 ANSI C++콤파일러들의 일부 제한성들을 극복하는 방법에 대하여 서술한다. 이것은 가능한 ANSI구조들을 유산콤파일러에 의하여 실현되는 이전에 쓰던 언어의 구성요소들에로 변환하는것에 의하여 실현된다. 이 변환처리는 보통 마크로처리기만을 사용해 가지고서는 실현할수 없다. 어떤 경우에는 ANSI C++원천코드의 수동적인 편집이 요구될 때도 있다.

26.1 개 요

이 장의 목적은 ANSI표준을 지원하는 콤파일러와 유산콤파일러사이의 차이점을 밝히는것이다. 그러나 여기서는 C++가 오랜 기간에 걸쳐 발전해 왔다는것을 고려하여 그의 초기시기의 차이에 대해서는 언급하지 않는다.

26.2 포함지령

ANSI표준에서 C++머리부파일들은 확장자를 가지지 않는다. 실례로 머리부파일 iostream의 포함(include)은 다음과 같다.

```
#include <iostream>
```

C++의 초기판본들에서 머리부파일은 명백히 .h확장자를 가지였다. 이러한 형식의 include지령을 지원하지 않는 유산콤파일러에 대하여 머리부파일 iostream의 포함은 다음과 같다.

```
#include <iostream.h>
```

26.3 실현되지 못한 론리형

프로그람의 웃부분에

```
#define bool int
#define true  1
#define false 0
```

인 마크로들을 정의하면 아래와 같이 명백한 코드를 쓸수 있다.

```
enum bool { false,true };
```

그러나 이 코드는 모든 경우에 대하여 동작하지 못한다.

26.4　for순환에서 시작값명령문의 유효범위

for명령문에서 시작값명령문이 있다면 그때 선언된 객체의 유효범위는 for명령문의 끝까지이다. C++언어의 이전 판본들에서 시작값변수의 유효범위는 그 변수를 둘러싼 블로크의 끝까지였다.

낡은 유효범위규칙을 가진 콤파일러들에 대해서는 괄호 {}로 for명령문전체를 둘러 싼다. 실례로

```
{
   for ( int i = 0; i < 10; i++ )      // 시작값명령문→ int  i=0
   {
         // 첫번째  for순환본체
   }
}
{
   for ( int i = 0; i < 10; i++ )     // 시작값명령문→ int  i=0
   {
         //두번째  for순환본체
   }
}
```

는 for순환본체에서 선언 int i=0의 유효범위를 효률적으로 제한한다.

26.5　new에 의한 동적기억기할당

다음의 형식을 리용한 new호출은 기억기를 더이상 할당할수 없을 때 레외 bad_alloc를 발생시킨다.

```
char  *p_chs = new char[ 10 ];
```

그런데 이전의 판본들에서는 기억기가 할당될수 없을 때 NULL지적자를 돌려 주었다. 이러한 이전 판본들에서의 효과를 얻자면 new를 다음과 같이 쓰면 된다.

```
char  *p_chs = new char ( nothrow ) [10];
```

26.6　실현되지 못한 레외기구

레외기구를 완전히 서술하기는 힘들다. 표준적인 레외들이 사용되였던 경우들에 쓰인 중간해결은 다음과 같다.

```
#define throw
#define try
#define catch(parameter) exception err; if ( false )
```

주의: 매 블로크안에는 하나의 례외포착만이 있을수 있다.
례외클라스들이 정의되여야 한다.

이것은 적어도 코드에 대한 콤파일이 수행되게 하며 례외가 발생되지 않는 조건에서 정확히 실행된다.

26.7 실현되지 못한 례외클라스

례외클라스를 다음과 같이 정의한다.

```
class exception
{
public:
    exception ( const string& arg = " " )
    {
        the_message = arg;
    }
    virtual ~exception( ) { };
    virtual const char* what( )
    {
        return the_message.c_str( );
    }
private:
    string the_message;
};
```

그때 사용된 매 례외에 대하여 그에 대한 례외클라스 exception로부터 파생클라스가 제공된다. 실례로 례외 logic_error에 대하여 다음의 클라스가 정의된다.

```
class logic_error : public exception
{
public:
  logic_error( const string& arg ) : exception( arg ) {};
};
```

26.8 실현되지 못한 이름공간지령

서로 다른 이름공간(namespace)들의 효과를 이미 쓰던 구조(construct)들을 리용하여 실현하기는 어렵다. 유일한 해결방도는 그 지령들에 설명문(comment)을 주고 그 이름공간의 매 이름에 그 이름공간의 이름을 앞붙이로 주며 그 새 이름을 반영할수 있도록 코드를 수정하는것이다. 이것은 일정한 시간이 걸린다. 이 책에서 본 실례들에서 다음의 마크로는 표준서고클라스사용에서 이름공간 std앞붙이를 삭제한것이다.

```
#define std
```

26.9 실현되지 못한 본보기기구

C++의 초기발전시기에는 본보기구조가 마크로를 리용하여 쉽게 모의되였다. 본보기들의 이러한 모의는 범용클라스에 대하여 마크로를 정의하는것으로 실현된다. 마크로는 그 클라스가 구체례제시(instantiation)될수 있는 형을 자기의 파라메터로 가진다. 마크로는 그때 그 클라스의 어떤 구체례제시를 선언하는데 리용된다.

실례로 수들을 차례로 발생시키는 발생기를 정의하는 본보기클라스는 다음의 마크로들을 사용하여 실현된다.

```
#define generate_seq( Type )                                          \
class generate_seq_##Type                                             \
{                                                                     \
public:                                                               \
    generate_seq##Type( Type initial_val , Type inc )                 \
          : the_next( initial_val ) , the_increment( inc )            \
    {                                                                 \
    };                                                                \
    Type operator() ()                                                \
    {                                                                 \
          Type res = the_next;                                        \
          the_next = the_next + the_increment;                        \
          return res;                                                 \
    }                                                                 \
private:                                                              \
    Type the_next;                                                    \
    Type the_increment;                                               \
};                                                                    \
```

주의: 어떤 기호의 부분을 대용할수 있도록 하는 ##를 리용한다.
이것은 TYPE가 generate_seq_## Type를 위한 새 기호를 만들기 위하여 어떤 형으로 대용될수 있도록 하는데 필요하다.

```
#define declare( Object, Type ) Object( Type )
```

로 정의된 마크로 declare는

```
declare( generate_seq, int )
```

와 같이 어떤 클라스 generate_seq_int를 구체례제시하는데 리용되며

```
#define implement( Object, Type ) Object##_##Type
```

로 정의된 마크로 implement는

```
implement( generate_seq, int )   int_seq(1,1);
```

과 같이 클라스 generate_seq_int의 구체례 init_seq를 실현하는데 리용된다.

주의: implement와 declare마크로들은 머리부파일 <generic>안에 정의되여 있다.

26.9.1 종합서술

이것들은 프로그람에서 다음과 같이 리용될수 있다.

```
#include <iostream>

declare( generate_seq, int )
declare( generate_seq, float )

int main()
{
    implement( generate_seq, int )                      int_seq(1,1);
    implement( generate_seq, float )       float_seq(10.2, 0.1);

    std::cout << "Next int is    : " << int_seq()  << " \n ";
    std::cout << "Next int is    : " << int_seq()  << " \n ";
    std::cout << "Next float is  : " << float_seq() << " \n ";
    std::cout << "Next float is  : " << float_seq() << " \n ";
    return 0;
}
```

실행결과는 다음과 같다.

```
Next int is         : 1
Next int is         : 2
Next float is       : 10.2
Next float is       : 10.3
```

26.10 실현되지 못한 mutable수식자

다음의 마크로는 프로그람의 웃부분에서 정의된다.

```
#define mutable
```

이때 const를 빈 문자렬(null string)로 정의하기 위해 mutable을 사용하는 클라스를 마크로로 둘러 싼다. 그러면 const속성이 함수명세부에서 제거되므로 검토자성원함수를 변이자성원함수로 볼수 있다.

```
#define const

class Uses_mutable { };

fundef const
```

이것은 클라스안에 다중정의연산자의 고정판본과 비고정판본이 없다는것을 가정한다.

26.11 실현되지 못한 explicit수식자

다음의 마크로는 프로그람의 웃부분에서 정의된다.

```
#define explicit
```

이것은 프로그람에서 수식자를 삭제한다.

26.12 클라스안의 초기화되지 않은 상수성원

렬거를 리용하여 값을 정의한다. 실례로 클라스의 비공개부안에서 MAX가

```
private:
    static const int MAX = 100;
```

와 같이 선언되면 이것을 다음과 같이 변경할수 있다.

```
private:
    emum { MAX  100 };
```

26.13 자체평가

- 유산콤파일러를 위하여 ANSI C++기능을 준비하는것을 해당한 마크로를 리용하여 어느 정도에까지 자동화할수 있는가?

26.14 련 습

- 자기가 리용하는 C++판본에서 실현되지 않는 ANSI C++의 기능들을 실현하기 위한 C++의 전처리기를 작성하시오.
- C++언어에 대한 새로운 확장을 실현하시오.

27 속 성

이 장에서는 C++항목들의 속성과 실행기간 그것들이 실현되는 방법에 대하여 서술한다.

27.1 소 개

C++에서 항목의 속성들은 복잡하며 그 속성들을 모든 경우에 대하여 일반적으로 서술할수는 없다. 이러한 조건에 비추어 볼 때 변수는 다음의 속성들을 가진다고 말할수 있다.

- 지속기간(duration)
- 련결(linkage)
- 유효범위(scope)
- 기억기클라스(storage class)
- 형(type)
- 보기가능성(visibility)

이것은 일부는 명시적이고 일부는 암시적인 수많은 인자들에 의해 확장된다.

27.2 수 명

수명(lifetime)은 항목의 활동지속기간(active duration)을 말한다. 그것은 다음 것들중에서 어느 하나에 해당된다.

- 프로그람의 수명을 위한것(static)
- 실행함수의 수명을 위한것(auto)
- 프로그람작성자에 의하여 결정되는것(dynamic)

실례로 다음의 프로그람은 정적, 자동, 동적수명의 특징을 보여 준다.

```
#include < iostream >
nt blue_cars = 3;
nt main( )
{
   int   red_cars = 4;
   int* green_cars = new int [ 1 ]; *green_cars = 5;
   std::cout << " Cars: Red        = " << red_cars << " \n ";
   std::cout << " Cars: Blue       = " << blue_cars << " \n ";
   std::cout << " Cars: Green= " << *green_cars << " \n ";
   delete green_cars;
   return 0;
}
```

주의: static와 auto수명들은 암시적으로 정의된다.

정적수명: blue_cars의 기억기는 정적지속기간을 가진다. 다시 말하여 그 기억기가 프로그람수명기간 유지된다. 함수밖에서 선언된 변수들은 항상 정적지속기간을 가진다.

자동수명: red_cars기억기는 자동지속기간을 가진다. 다시 말하여 그 기억기가 그 함수의 수명기간 유지된다. 이 변수의 기억기는 실행시 탄창안에 할당된다.

동적수명: 프로그람작성자는 연산자 new로써 green_cars를 위한 기억기를 명시적으로 할당하며 delete로써 그것을 해제한다.

우의 프로그람은 다음과 같이 auto와 static의 명시적인 선언으로 작성할수 있다.

```
#include<iostream>
static int blue_cars = 3;
int main( )
{
    auto int red_cars    = 4;
    int* green_cars = new int [ 1 ]; *green_cars = 5;
    std::cout << " Cars: Red        = " << red_cars << " \n ";
    std::cout << " Cars: Blue       = " << blue_cars << " \n ";
      std::cout << " Cars: Green= " << *green_cars << " \n ";
    delete green_cars;
    return 0;
}
```

주의: 함수의 국부변수들은

static int red_cars;

와 같이 static의 뒤에 붙을수 있는데 이 경우에 변수는 대역자료기억기에 할당되며 프로그람수명기간 유지된다. 이것은 함수들이 값을 설정된 시각에 실행할수 있게 한다. 프로그람작성에서는 대체로 함수의 국부변수들이 자동지속기간(auto duration)을 가지게 하는것이 좋다.

27.2.1 요약

선언된 항목	기정의 지속기간 (선언이 요구되지 않음)	기타 (선언이 요구됨)
함수안에서	자동	정적
함수밖에서	정적	

27.3 련결

여러개로 분할콤파일된 모듈들을 리용하여 프로그람을 작성할 때 어떤 모듈에서 선언된 항목들이 다른 모듈에서 리용되는 경우가 있을수 있다. 련결(linkage)은 이러한 련관이 실현되도록 하는 속성이다. 모든 항목들은 다음의 속성들을 가진다고 말할수 있다.

- 비련결
- 내부련결　　static int not_visible_outside
- 외부련결　　extern int visible_outside

주의: 기정적으로 함수들은 외부련결을 가지며 변수들은 내부련결을 가진다.

다음의 프로그람은 련결을 실현한다. 2개의 프로그람 ex1.cpp와 ex2.cpp는 공통적인 머리부파일 ex.h를 리용하여 따로따로 콤파일된다.

```
//머리부파일 ex.h
# ifndef HEADER_FILE_EX
# define HEADER_FILE_EX
extern int shared;              // shared는 외부련결을 가진다
void process( );                // 처리를 위한 함수원형
# endif
```

프로그람파일 ex1.cpp	프로그람파일 ex2.cpp

# include " ex.h " 	
 	
int shared;
int main( )
{
    shared = 42;
    process ( );
    return 0;
}
	# include " ex.h "
	# include < iostream >
	
	void process (void)
	{
	    std::cout << " Shared = ";
	    std::cout << shared << " \n ";
	    return;
	}

2개의 객체코드파일들은 함께 련결되며 실행할 때 결과프로그람은 다음의 결과를 인쇄한다.

```
Shared = 42
```

주의: 함수 process는 기정으로 외부련결을 가진다.
공통머리부파일은 shared가 외부련결을 가져야 한다는것을 콤파일러에 지시한다.
파일 ex1.cpp안에서 콤파일러는 목적코드안에 ' I am here' 참조를 발생시킨다.
파일 ex2.cpp안에서 콤파일러는 목적코드파일안에 'link to' 참조를 발생시킨다.

27.4 유효범위

C++프로그람에서 유효범위(scope)는 항목이 능동으로 되는 기억령역을 결정한다. 어떤 경우에 이것은 항목이 참조될수 있는 기억령역과 같지 않을수 있다.

다음의 프로그람을 보기로 하자.

```
std::string var_is_a = " Global declaration ";
int main( )
{
    std::string var_is_a = " Local declaration ";
    {
          std::string var_is_a = " Local declaration ";
          std::cout << " var_is_a is : " << var_is_a << " \n ";     // 국부의 내부
          std::cout << " var_is_a is : " << ::var_is_a << " \n ";   // 대역
    }
    std::cout << " var_is_a is : " << var_is_a << " \n ";           // 국부
    return 0;
}
```

함수 main에서 var_is_a의 첫번째 선언은 함수 main의 지속기간에 대하여 유효범위안에 있지만 두번째 선언이 이루어 지는 블로크의 지속기간에는 숨겨 진다. 유효범위해결연산자 ::는 대역적인 var_is_a에 접근하는데 리용된다.

유효범위에는 4가지 종류가 있다.

- 클라스유효범위
- 파일유효범위
- 함수유효범위
- 국부적유효범위

클라스유효범위: 클라스안에서 선언된 항목들은 그 클라스에 대하여 국부적이다. 클라스성원들이 비공개부접근권이나 보호부접근권에 의하여 숨겨 지는 조건에서는 연산자 ::나 . 혹은 →를 리용하여 그 클라스성원들에 대한 접근이 진행된다.

파일유효범위: 클라스나 함수, 블로크의 밖에서 선언되는 항목은 선언된 곳에서 콤파일단위의 끝에 이르는 기간 볼수 있다.

함수유효범위: 함수안에서 선언된 모든 표식들은 함수의 임의의 점에서부터 호출될수 있다. 표식은 go to명령의 목적지이다.

```
void nasty ( const int data )
{
   // 이름표 exit는 함수의 유효범위안에 있다
    if ( data < 10 ) goto exit;
exit:
    return;
}
```

국부적유효범위: 함수안에서 선언된 항목은 선언된 곳에서 블로크의 끝까지에 이르는 블로크({}로 둘러 막힌)에 대하여 국부적이다.

27.5 보기가능성

보기가능성은 항목이 접근될수 있는 기억령역을 정의한다. 이것은 우에서 본 var_is_a의 경우에서와 같이 항목의 유효범위보다 더 적을수 있다.

C++에서 선언된 매개 형이름은 같은 이름공간을 공유한다. 만일 같은 이름을 가지는 2개의 서로 다른 형이 있으면 반드시 형을 언급해야 하는 유일동등화방법이 있어야 한다.

27.6 기억기클라스

기억기클라스는 항목이 자기의 수명기간에 기계안의 어디에 어떻게 기억되는가를 결정한다.

쓸모 있는 기억기클라스들은 다음과 같다.

- 자동　auto　국부변수
- 외부　extern　정적변수
- 등록기　register　국부변수
- 정적　static　프로그람이 존재하는 동안
- 변이　mutable　클라스자료성원을 쓰기가능

자동: auto는 함수에서 국부적인 항목들만 가지고 리용될수 있으며 항목이 실행시 탄창안에 기억된다는것을 가리킨다.

```
void process( ) {
    auto int temp;
}
```

외부: extern은 외부적인 련결이 extern항목에 대하여 만들어 진다는것을 가리킨다. 명백히 이러한 항목의 위치는 반드시 콤파일시에 알려 져야 하며 따라서 암시적으로 정적이여야 한다.

등록기: register는 가능하다면 CPU등록기안의 항목을 가지는 콤파일러에 대한 암시이다. 해당크기와 자동으로 정의된 항목들만이 이러한 취급에 적합하다. 콤파일러는 등록기안의 항목위치에 대한 암시를 무시할수 있다. C++에서는 기억기클라스등록기를 가지는 항목의 주소를 가질수 있다. 이에 대하여 가장 있음직한 효과는 콤파일러가 register안의 항목위치에 대한 암시를 무시한다는것이다.

```
void process( )
{
```

```
    register int used_frequently;                    // CPU등록기안에 들어 있다
}
```

정적: static는 항목이 내부적련결을 가지며 그의 수명은 프로그람의 수명과 같다는것을 의미한다.

```
int new_product_code( )
{
    static int number =0;  // 유일한 결과를 넘겨 준다

        return ++number;  //호출될 때마다 유일한 결과를 넘겨 준다
}
```

변이: mutable은 메쏘드에 대한 const지정자를 무효화하기 위한 클라스의 자료성원에 적용된다. 실례로 const메쏘드(검토자)는 클라스의 변이자료성원을 변경시킬수 있다.

```
class Bank
{
public:
    float account_balance( const int ) const;
private:
    mutable fstream the_customers;
}
```

검토자메쏘드 account_balance는 그것이 디스크에서 개별적사용자의 세부를 읽을 때 흐름지적자 the_customers를 변화시킨다. 그러나 이 메쏘드 account_balance는 그것이 파일을 변경시키지 않으므로 고정적(클라스의 의뢰자에 대하여)이며 변이메쏘드로 되는 경우 의뢰자에게 혼돈을 일으킨다. mutable은 주의하여 사용하여야 한다.

27.7 변경자

선언은 다음과 같은것들에 의하여 변경될수 있다.

const: 항목을 읽기만 할수 있다는것을 가리킨다.

```
const int MAX = 10;                            //항목들의 최대수
```

주의: 클라스의 구체례는 const로 선언될수 있는데 이때에는 고정객체에 관하여 작용하는 함수들도 반드시 const로 선언되여야 한다.

volatile: volatile은 항목이 그 프로그람과 다른 원천에 의하여 변경될수 있다는것을 콤파일러에 경고한다. 실례로 입출력기억기를 사용한 장치등록기는 표준적인 기억위치로서 나타난다. 이 기억위치에 접근하는 프로그람은 콤파일러가 CPU등록기에서 그의 값을 가리우든가 아니면 그 코드를 최량화하도록 해야 한다.

```
struct Acia {
        char    status;                         // 상태정보
        char    dummy1;                         // 1바이트여유
        char    data;                           // 자료등록기
};
volatile Acia * base_vdu = reinterpret_cast<Acia*>( 0x0c0080 );
const int ACIA_RM     = 0x01;
const int ACIA_RM     = 0x02;
// 장치가 해방될 때까지 기다린다
// 말단장치로부터 한 문자를 읽는다

char getchar( )
{
   while( ( bast_vdu -> status & ACIA_RM ) == 0 );
   return base_vdu -> data;
}
```

주의: 클라스의 구체례는 volatile로 선언될수 있다. 그때에는 이 객체에 보내진 통보문들이 volatile메쏘드를 요구해야 한다.

27.8 형

선언될수 있는 형에는 여러가지 부류가 있다.

- 집합체형(aggregate)
- 함수형(function)
- 기본형(fundamental)
- 빈 형(void)

27.8.1 집합체형

집합체형들은 기본(fundamental)형들로부터 파생된다. 이러한 집합체형들은 다음과 같다.

- 배렬 int vec[4];
- 클라스 class Account { };
- 지적자 char *p_ch;

- 참조　　　char& passed;
- 구조체　　struct   Person { };
- 공용체　　union   Overlay { };

27.8.2 함수형

함수형은 프로그람안에서 실행될수 있는 코드렬을 정의한다.

27.8.3 기본형

기본(fundamental)형들의 구체례는 보통 단일한 기계에 기초한 구성요소안에 포함되며 산수형으로 알려 져 있다.

- 옹근수형들: enum,char와 int의 모든 크기
- 류점수: float,double, long double

27.8.4 빈 형

빈 형(void)은 어떤 값이 없다는것을 지적한다. 이것은 기본적으로 다음과 같은 것에 사용된다.

- 함수가 돌림값을 가지지 않는다는것을 지적한다.
- 함수가 파라메터를 가지지 않는다는것을 지적한다.
- 어떤 형의 기억기에 대한 지적자를 서술한다.

실례로

```
void hello( );                // 함수원형
void * not_sure;              // 어떤 형의 기억기에 대한 지적자
void hello( ) { std::cout << " Hello " << " \n ";}
```

27.9 프로그람의 실행시 집행

C++프로그람의 실행은 다음과 같다.

```
# include < iostream >
typedef char* C_string;
c_string str_letters( int );
int length = 10;
int main( )
{
    C_string letters = str_letters( length );
```

```
    std::cout << " The first " << length;
    std::cout << " letters are " << letters << " \n ";
    delete [ ] letters;
    return 0;
}
c_string str_letters( int len )
{
    c_string text = new char[ len + 1 ];
    for ( int i =0; i < len; i++ ) text [ i ] = char ( i + int ( ' a ' ) );
    text[ len ] = ' \0 ';
    return text;
}
```

실행결과는 다음과 같다.

```
The first 10 letters are abcdefghij
```

C++프로그람의 실행을 지원하기 위해 항목들은 기억기의 서로 다른 기억령역안에 할당된다. 비록 실현부에서는 약간의 차이가 있을수 있더라도 포함된 기억령역들은 그림 27-1에 보여 준다.

	령역	C++ 항목들
높은 준위 기억기	실행시 탄창	letters;
	더미	new char[len+1]
	대역자료	length;
낮은 준위 기억기	코드	main str_letters

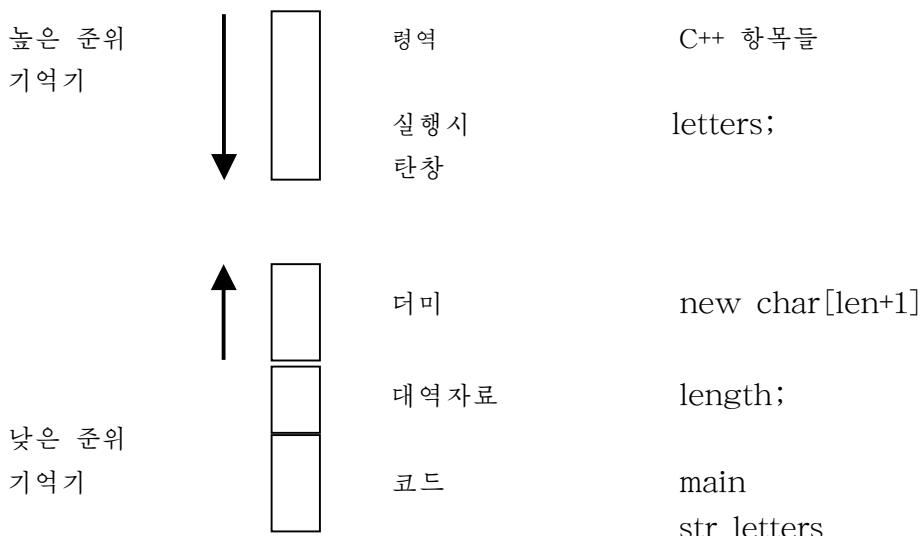



그림 27-1. 기억기배치

27.9.1 실행시 탄창

실행시 탄창은 개별적인 탄창틀(stackframe)을 포함하는데 이것은 C++프로그람에서 호출된 함수들의 실행을 지원한다. 함수가 호출될 때마다 새 탄창틀이 만들어 진다.

실례로 함수 str_letters가 호출될 때 그의 탄창틀은 그림 27-2에서 설명한것처럼 된다.

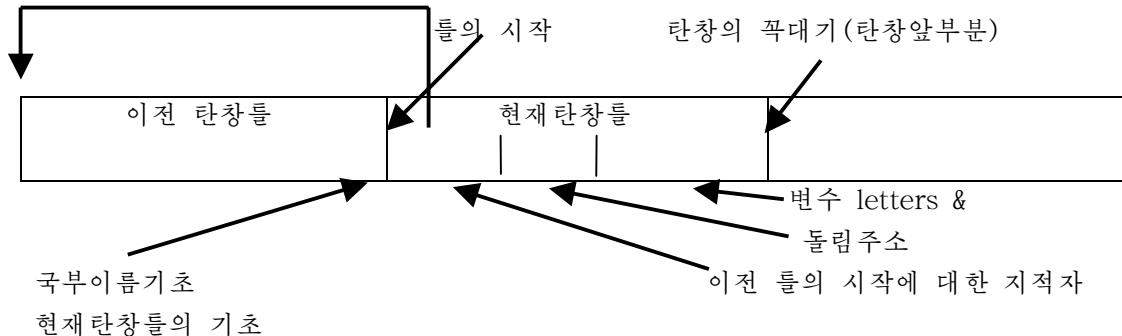



그림 27-2. C++프로그람의 실행시 탄창

그림 27-2에서

- LNB(Local Name Base:국부이름기초)는 지적자로 리용되는데 현재탄창틀안의 항목들은 LNB에서부터 접근될수 있다. 탄창틀은 현재국부변수들을 포함한다.
- TOS(Top Of Stack:탄창꼭대기)는 틀의 웃한계를 표시한다.

str_letters를 위한 탄창틀안의 항목들 ;

이전 탄창틀의 지적자 :

이것은 현재함수로부터의 탈퇴가 진행될 때 능동LNB로 된다.

돌림주소 :

이것은 현재함수로부터의 탈퇴가 진행될 때 실행을 다시 시작하게 하는 코드주소를 가리킨다.

변수 letters와 i:

함수안에서 선언된 auto변수들은 탄창틀안에 할당된다. 따라서 함수가 탈퇴될 때 이 기억공간은 해방되여 체계에 넘겨 진다.

림시적인 작업공간:

함수의 코드를 평가하는데서 림시값들은 탄창의 앞에서 탄창틀을 확장하는것에 의하여 기억될수 있다.

27.9.2 더미령역

new로 할당된 기억기는 그것이 delete로써 명시적으로 해방될 때까지 존재한다. 이 기억기는 더미령역(heap)으로부터 이루어 지는데 이것은 돌려 진 기억기를 다시 할당해 주는 불필요한 기억령역수집기(garbage collector)에 의해 관리된다. 만약 재할당될수 있는 기억기가 없다면 실행시 탄창쪽으로 더미령역을 확장하는것에 의해 새 기억기가 할당된다.

좋기는 실행시 탄창과 더미령역을 쓰지 않는것이다.

주의: 이것은 보통 기억기관리단위에 의하여 실시된다. 그러나 일부 실행상에서 이것이 검사되지 않고 2개의 령역이 서로 합쳐 져 범위를 넘으면 실패를 가져 올수 있다.

27.9.3 대역자료

대역자료령역(global data area)은 보통 정적으로 할당된 모든 자료항목들이 기억되여 있는 령역이다. 일반적으로 이것은 함수의 밖에서 선언된 자료항목들이다.

27.9.4 코드

코드령역은 프로그람의 물리적인 기계코드를 포함한다.

28 C++에 대한 개요

이 장에서는 C++언어의 주요기능들을 개괄적으로 서술한다.

C++에서 선언들

```
char      c;      unsigned char c;
wchar_t   wc;
bool      b;
int       i;      unsigned int   i;     signed int            i;
                  short int      i;     unsigned short int    i;
                                        signed short int      i;
                  long int       i;     unsigned long int     i;
                                        signed long int       i;
float     f;      double         f;     long double           f;

Account  mike;
```

렬거선언

```
enum Color { RED, GREEN, BLUE };

Colour  car = RED;
```

배렬선언

```
char text [ 120 ];                // 120개 문자들의 배렬
char mes [ ] = "Hello world";     // 배렬
char table [ 3 ] [ 3 ];           // 2차원배렬
```

동적기억기할당

```
int main( )
{
   float *p_f = new float [ 1 ];         // 기억기할당
   *p_f = 12.3;    std::cout << *p_f;    // 리용
   delete [ ] p_f;                       // 할당해제
}
```

수명

```
char  global;                // 이 프로그램이 존재하는 전 기간 볼수 있다

int main( )
{
    char l;                  // 함수의 국부변수
    static char c;       // 이 프로그램이 존재하는 기간 함수안에서만 볼수 있다
    char *d = new [ 10 ];    // 동적할당
    delete [ ] d;            // 할당해제
}
```

클라스의 명세부와 실현부

```
class Account {
public:
    Account( );
    float account_balance ( ) const;         // 잔고를 돌려 준다
    float withdraw ( const float );          // 구좌에서 출금한다
    void deposit (const float );             // 구좌에  저금한다
protected:
    void set_min_balance ( const float );    // 최소잔고를 설정한다
private:
    float the_valance;                       // 현재잔고
    float the_min_balance;                   // 최소잔고
};
```

```
Account::Account( ) { the_amount = the_min_balance = 0.00; }

float Account::account_balance( ) const { return the_balance; }
```

```
void  Account::deposit ( const float  money )
{
    the_balance = the_balance + money;
}
```

```
float Account::withdraw ( const float  money )
{
    if ( the_balance - money >= the_min_balance )
    {
          the_balance = the_balance - money;
          return  money;
    } else {
```

```
        return  0.00;
    }
}
```

```
void Account::set_min_balance ( const float money )
{
    the_min_balance = money;
}
```

계승

```
class Interest_Account : public Account
{
    public;         // 구좌를 지정하는 함수들
    protected;      // 구좌를 지정하는 함수들
    private;        // 구좌를 지정하는 구체레변수들
};
```

다중정의연산자와 값주기

```
class List {
public:
    List ( );                       // 구축자
    ~List ( );                      // 해체자
    List ( List& )                  // 복사구축자
    List operator = ( List );       // 값주기연산자
    List operator + ( List );       // 더하기연산자
    operator int ( );               // 변환연산자
private:
};
```

실행시 형의 일치

```
int main( )
{
    Account  mike;
    std::cout << " Type of mas = " << typeid ( mike ) . name ( ) << " \n ";
    return 0;
}
```

강제형변환

```
int main( )
{
    float i = ( float ) 2;
    float j = float ( 2 );                                      // 함수적인 표시
    Account *p = new Restricted_account [ 1 ];
    Restricted_account *r = dynamic_cast <restricted_account*>( *p ); // 내리변환
    return 0;
}
```

함수

```
int twice( int );                                  // 함수원형
int main( )                                        // 입장점
{
    int actual_param = 2;
    std::cout << " Twice actual is " << twice ( actual_param );
    return 0;
}
int twice( int  formal_param )                     // 함수본체
{
    return  formal_param + formal_param;
}
```

명령식

```
a = 2 + 3;                    // 값주기식
bank_statemen ( );            // 함수호출
mas.deposit ( 10 );           // mas에 10을 저금하는 통보문을 보낸다
```

복합명령문

```
{
    char character;
    character = 'M ';  std::cout << character;
}
```

선택명령문

```
if ( temperature < 15 )  std::cout << "Cold";
if ( temperature < 15 )
    std::cout << "Cold";
else
    std::cout << " Warm ";
switch ( number )
{
    case 2 + 3    :  std::cout << " Is 5 ";
                     break;
    case 7        :  std::cout << " Is 7 ";
                     break;
    default       :  std::cout << " Not 5 or 7 ";
}
std::cout << ( temperature < 15 ? " Cold " : " Warm " );
```

순환명령문

```
while ( raining )  work( ) ;
do
    play ( );
while ( sunny );
for ( i = 1; i < MAX; i++ )  std::cout << i << " \n ";
```

산수연산자

```
res = a + b;      // 더하기
res = a - b;      // 덜기
res = a * b;      // 곱하기
res = a / b;      // 나누기(a와 b가 다 옹근수이면 결과는 옹근수)
res = a % b;     // 모듈러연산(나머지연산)
```

조건식

```
if ( a ==b )   // 같기
if ( a > b )   // 크기
if ( a < b )   // 작기
if ( a != b )  // 같지 않기
if ( a >=b )   // 같거나 크기
if ( a <= b )  // 같거나 작기
```

```
if ( wet && monday )  // 론리곱하기
if ( dry || tuesday )      // 론리더하기
```

```
if ( cost > 15 && number <= 100 )
{
   std::cout<<"Cost greater than 15 and the number of"<<"items is less than 100";
}
```

주의: 조건식은 조건결과를 얻는데만 필요하게 평가된다. 다음의 if명령문에서 함수 fun_two( )은 함수 fun_one( )이 참을 넘겨 주면 호출되지 못한다.

```
if ( fun_one( ) || fun_two( ) )  perform ( );
```

론리연산자

```
res = a << 2;       // 2진수 두자리만큼 왼쪽 밀기
res = a >> 2;       // 2진수 두자리만큼 오른쪽 밀기
res = a & b;        // a와 b의 론리곱하기(and)
res = a | b;        // a와 b의 론리더하기(or)
res = a ^ b;        // a와 b의 배타적론리더하기(xor)
res = !a;           // a의 론리부정(not)
res = ~a;           // a의 1의 보수
```

간략법

```
a ++;                    // a=a+1;과 같다
a --;                    // a=a-1;과 같다
res += a * b;            // res=res+(a*b)와 같다
                         // 기타 -=, *=, /=, %= 등
```

순환안에서 조종흐름의 변경

```
while ( raining )
{
    work ( );
    if ( tied )  break;                    // 만일 tied이면 순환에서 탈퇴
    work ( ) ;
}

while ( raining )
{
    work( );
    if ( may_have_stoped )  continue;      // 순환의 시작에로 뛰여넘기
    work( );
}
```

례외

```
try {
     // 례외를 발생시킬수 있는 코드
    throw std::runtime_error(“It want wrong”);
}
catch ( std::runtime_error& err )
{
    std::cout << " Problem " << err.what ( ) << " \n ";
}
```

주소연산자

```
chat string [ ] = " Hello world ";
char *p_ch;                    // p_ch는 char형지적자를 가진다
p_ch = &string [ 0 ];    // p_ch에 문자렬의 첫번째 요소의 주소를 값주기한다
```

다음의 코드는 p_ch에 의하여 지적되는 문자렬안의 문자들을 인쇄한다.

```
while ( *p_ch ) std::cout << *p_ch++;
```

함수본보기

```
template <class Type>
Type max ( Type first , Type second )
{
    return first > second ? first :second;
}
```

```
int main ( )
{
    std::cout << max ( 4.64 , 3.14 ) << " \n ";
    std::cout << max ( 'M ', 'S ' ) << " \n ";
    return 0;
}
```

클라스본보기

```
template <class Type, const int MAX_ELEMENTS = 5>
class Stack {
```

```
public:
  Stack();
  void push ( const Type );                    // 탄창에 항목넣기
  Type pop();                                  // 탄창에서 항목꺼내기
  bool empty();                                // 빈 탄창
private:
  Type the_elements[MAX_ELEMENTS];             // 탄창의 항목들
  int the_tos;                                 // 탄창꼭대기지적자
};
```

```
template <class Type, const int MAX_ELEMENTS>
Stack < Type, MAX_ELEMENTS > :: Stack
{
  the_tos = -1;                                // 빈
}
```

```
int main()
{
  stack < int, 10 > numbers;                   // 최대길이가 10인 옹근수탄창
  stack < float > values;                      // 최대길이가 5인 류점수탄창
}
```

부록 1. C++형식의 입출력

1) 기정흐름

std::cout	표준적으로 말단화면
std::cin	표준적으로 말단건반
std::cerr	이 흐름은 항상 사용자말단에 출력을 보내는것이 일반적이다.

주의: std::cout , std::cin , std::cerr는 std::osteream형이다.

2) 출력조작자

출력조작자는 마치 출력되여야 할 일반자료항목처럼 쓰인다. 그러나 출력되는 자료가 아니라 출력되여야 할 자료가 어떤 형식을 가지겠는가를 결정하고 설정한다. 실례로 수 42를 인쇄하는 마당너비를 5자리로 설정하기 위하여서는 다음과 같이 쓸 수 있다.

```
std::cout << std::setw(5) << 42;
```

출력조작자를 리용하기 위하여서는 프로그람에 머리부파일 iomanip.h를 포함하여야 한다. 즉 #include <iomanip>라고 써야 한다.

조작자	항목	효 과
std::setw (int n)	다음항목	다음항목이 n개의 마당폭에 인쇄되도록 마당너비를 설정한다.
std::hex	모든 항목	모든 련속적인 옹근수항목들을 16진수로 출력하도록 설정한다.
std::oct	모든 항목	모든 련속적인 옹근수항목들을 8진수로 출력하도록 설정한다.
std::dec	모든 항목	모든 련속적인 옹근수항목들을 10진수로 출력하도록 설정한다.
std::setfill(int c)	다음항목	다음항목을 마당폭에 인쇄하되 빈 자리에는 c문자로 채운다.
std::setprecision(int n)	모든 항목	류점수의 소수부자리를 n개로 설정한다.
std::endl	다음항목	행바꾸기를 한 다음 완충기억기의 흐름내용을 모두 비운다(flush).
std::flush	다음항목	완충기억기의 문자흐름내용을 모두 비운다.
std::ends	다음항목	문자렬끝에 기호 "\n"을 삽입한다.
std::boolalpha	모든 항목	문자형식화에 론리형을 삽입한다.
std::uppercase	모든 항목	소문자를 대문자로 교체한다.
std::unitbuf	모든 항목	매 연산을 집행한후 완충기억기의 출력을 비운다.

주의: 두번째 렬은 그 결과가 모든 항목을 출력하기 위한것인가 혹은 조작자뒤의 항목을 출력하기 위한것인가를 가리킨다.

선택자 setiosflags와 resetiosflags의 파라메터

입출력선택자(ios)기발은 각각 조작자 setiosflags와 resetiosflags로서 설정 혹은 비설정된다. 실례로 모든 출력결과자료들이 선택된 출력마당폭의 오른쪽에 정렬되도록 설정하기 위하여 다음과 같이 쓸수 있다.

```
cout<<setiosflags (ios::right);
```

다음의 기발들은 클라스 ios에서 정의되며 조종자 setiosflags와 resetiosflags의 파라메터들로서 작용한다.

ios기발	항목	효 과
std::ios::left	모든 항목	출력마당의 왼쪽부터 항목들을 정렬시킨다.
std::ios::right	모든 항목	출력마당의 오른쪽부터 항목들을 정렬시킨다.
std::ios::scientific	모든 항목	류점수항목들을 어떤 지정된 표기법으로 출력한다. 실례로 42.2를 4.22E+01로 표시한다.
std::ios::fixed	모든 항목	류점수항목들을 소수표기법으로 출력한다. 실례로 42.2를 42.2로 표시한다.
std::ios::showpoint	모든 항목	류점수항목들을 소수표기법으로 출력한다. 이때 지정된 소수부자리에서 빈 자리는 령으로 채운다. 실례로 소수부가 2자리로 설정되였을 때는 4.2를 4.20으로 표시한다.
std::ios::showpos	모든 항목	정수들에 대해서는 +부호를 붙여 준다. 실례로 42를 +42로 표시한다.

실례로 코드부분

```
std::cout << “ The programming language [ ” ;
std::cout << std::setiosflags (ios::left);
std::cout << std::setw(3)<<std::setfill( ‘+’ ) << “ C ” << “ ] ” << “ \n ” ;
```

의 결과는 다음과 같다.

```
The programming language [C++]
```

3) 입력조작자

다음의 조작자들은 입력흐름에서 리용될수 있다.

조작자	항목	효　과
std::ws		입력에서 공백문자들을 읽는다.
std::oct,std::dec,std::hex	모든 항목	8 진수 혹은 10 진수, 16 진수로 입력한다.
std::boolalpha	모든 항목	문자형식에 론리형을 삽입한다.

선택자 setiosflags 와 resetiosflags 의 파라메터

이것들은 각각 조작자 std::setiosflags 와 std::resetiosflags 로 설정 혹은 비설정되는데 클라스 ios 에서 정의되는 다음의 기발들을 파라메터로 가진다.

ios 기발	항목	효　과
std::ios::skipws	모든 항목	입력연산들은 공백문자들을 무시한다. std::resetiosflags(std::ios::skipws)는 공백문자를 뛰여 넘는다.

4) 파일에 대한 입력과 출력

두개의 클라스 ofstream 과 ifstream 은 각각 ostream 과 istream 에서 파생된 클라스들이다. 이 클라스들은 파일에 대한 읽기와 쓰기를 할수 있다. 실례로 다음의 코드는 mas.txt 파일에 쓴 다음 그 자료를 사용자의 말단에 표시하면서 다시 읽는다.

```
#include<fstream>
#include <iomanip>
int main ( )
{
  std::ofstream sink;                                        // 출력흐름
  std::ifstream source;                                      // 입력흐름
  char c;
  sink.open ( “mas.txt ” );                                  // mas. txt 와 련결(열기)
  if (!sink)                                                 // 만들기실패
  {
    std::cerr << “Failed to create mas.txt ” << “ \ n ”;
    exit(-1);
  }
  std::sink << “ Hello world ” << “ \n ”;
  std::sink.close( );                                        // 흐름닫기
```

파일을 닫은 다음에는 그 파일을 입력원천으로 하여 다시 연다.

```
source.open(“ mas.txt”);                                     // mas. txt 와 련결(열기)
if (!source)                                                 // 만들기실패
{
```

```
    std::cerr<< “ Failed to open mas.txt ” << “ \n ” ;
    exit (-2);
  }
 source >>std::resetiosflags (std::ios::skipws);                    // 공백읽기
  while (source>>c,!source.eof( ) )                                 // mas.txt 를 cout 에 복사
  {
    cout<<c;
  }
  source.close( );                                                  //흐름닫기
}
```

이 프로그람의 출력결과는 다음과 같다.

```
Hello world
```

주의: 머리부파일 fstream.h 에는 fstream 과 ostream 의 정의가 들어 있다.

문자렬흐름

클라스 ostrstream 과 istrstream 은 각각 ostream 과 istream 에서 파생된 클라스들이다. 그러므로 이 클라스들은 기억구역의 읽기와 쓰기에 리용될수 있다. 실례로 프로그람

```
#include <strstream>
#include <iomanip>
#include <string>

const int MAX_MES = 100;                                          // 문자렬크기

int main ( )
{
  char mes [MAX_MES];
  char town [ ] = “Brighton East Sussex”;

  std::ostrstream ostr ( mes,  MAX_MES);                          // str 문자렬출력

  ostr << “ The sum of 2+3 is ” << (2+3)  << “ \n ” << ‘ \0 ’ ;
  std::cout <<mes;

  std::istrstream istr (town, strlen (town) );                     // str 문자렬입력
  istr >> resetiosflags (ios::skipws);

 char c;
  std::cout << “ \n ”  << “ [ ” ;

while ( istr >> c, !istr. eof ( ) )                               // 문자읽기
  {
    std ::cout << c;                                               // 문자렬흐름
```

```
  }
  std::cout << " ] " << " \n " ;
  return 0;
}
```

의 결과는 다음과 같다.

```
The sum of 2+3 is 5
[Brighton East Sussex]
```

주의: 일반적으로 C++문자렬에는 문자렬흐름에 문자렬끝기호 " \n "이 붙는다.
문자렬입출력을 위해 머리부파일 strstream.h 에 #include 를 붙인다.

부록 2. C 형식의 입출력

대부분의 C++콤파일러들은 C형식의 입력과 출력을 포함하고 있다. C++에서와 마찬가지로 C의 입출력루틴(routine)들은 사용자프로그람작성부분이 아니라 표준함수서고에서 제공된다. C에서 입출력체계는 safe형이 아니다.

C의 입출력방법에서는 파일과 결합되는 흐름지적자를 리용한다. 이 흐름지적자는 선택된 파일의 정보를 읽기 혹은 쓰기한다. 이 입출력체계에 속하는 선언들은 머리부파일 stdio.h에 들어 있다.

흐름지적자는 다음과 같이 선언된다.

```
FILE *sp;
```

그리고 서고함수 fopen 을 리용하여 파일과 결합시킨다. 즉

```
sp = fopen ( " file.dat ", " r " );
```

fopen 의 파라메터들과 돌림값들은 다음과 같다.

파라메터 1: 파일이름을 표시하는 문자렬.
파라메터 2: 파일의 접근방법을 표시하는 문자렬.
r: 읽기 위하여 연다.
w: 쓰기 위하여 만든다.
a: 파일이 있다면 그 파일뒤에 추가하기 위해 연다.
파일이 없다면 쓰기 위해 만든다.
r+: 읽기, 쓰기를 위해 파일을 연다.
w+: 새로운 파일 혹은 읽기, 쓰기를 위해 만든다.
a+: 파일이 있다면 그 파일뒤에 추가하기 위해 연다.
파일이 없다면 쓰기 위해 만든다.
결과: 파일에 대한 흐름지적자.
파일을 열수 없다면 NULL 을 넘긴다.

주의: 파일에 쓰기 위해 열거나 추가적인 접근을 할 때 그 파일이 없으면 먼저 파일을 만든다.

fopen 은 일반적으로 다음과 같이 호출된다.

```
sp = fopen ( “file.dat ” ,  “ r ” );
if (sp == NULL )
{
 /* 해당한 동작 즉 파일이 열려 지지 않는 경우의 처리내용을 서술한다 */
}
```

주의 : NULL 은 머리부파일 <stdio>에 정의되여 있다.

출력형식화

서고함수 fprintf 는 파일에 정보를 쓴다. 이 함수는 현재 있는 파라메터의 내용을 문자흐름에 어떻게 삽입하겠는가를 알리는 형식화문자렬을 리용한다. 이 함수를 리용하는 실례를 보기로 하자.

```
int main ( )
{
  FILE *sp;
  char  ch ;
  sp= fopen (  “ tmp.dat ” ,  “ w ” );
  for(int i = 100; i < 105;  i++)
  {
     fprintf (sp,  “ Character  %c  has  ASCII  code  %d  \n ” , i , i , );
   }
   fclose ( sp );
   return 0;
}
```

실행하면 그 결과가 tmp.dat 파일에 씌여 진다.

```
Character d has ASCII  code 100
Character e has ASCII  code 101
Character f has ASCII  code 102
Character g has ASCII code  103
Character h has ASCII  code 104
```

주의: fprintf 는 제공되는 인수들의 형에 대해서는 검사하지 못한다. 따라서 정확한 코드를 써야 한다.

fprintf 의 파라메터와 해당한 돌림값의 일부는 다음과 같다.

파라메터 1: 흐름지적자.

파라메터 2: 파라메터가 출력본문에 삽입된 다음 그 파라메터를 어떻게 표시하겠는가를 서술하는 문자렬. 삽입될 매 파라메터는 %다음에 형식화를 표시하는 문자나 문자들을 붙이는것에 의하여 소개된다.

%d 10 진수로서
%l 10 진수로서
%u 부호 없는 10 진수로서
%c 문자로서
%s 문자들의 렬로서(문자렬마감은‘\0’으로 되여 있음)
%f 류점수로서(형식 1.23)
%e 류점수로서(형식 1.23e45)
%o 8 진수로서
%x 16 진수로서
%ld 긴 옹근수로서
%1 긴 옹근수로서
%% %문자

형식들은 다음과 같이 변경될수 있다.

%4d 오른쪽으로부터 시작하여 4 개의 마당폭에 10 진수 출력
%_4d 왼쪽으로부터 시작하여 4 개의 마당폭에 10 진수 출력
%0.4d %4d 와 같으나 빈 자리는 0 으로 채운다.
%8.2f 오른쪽으로부터 시작하여 8 개 마당폭중 2 자리는 소수부자리에 류점수출력

결과: 조작의 성공 혹은 실패

다음의 흐름들은 보통 사용자를 위하여 미리 정의되여 있다.

stdout	표준적으로 말단화면
stdin	표준적으로 말단건반
stderr	표준적으로 말단화면. 이 흐름은 항상 사용자말단에 출력을 보내는것이 일반적이다.

한 문자출력

```
fputc (ch, stdout );            // 함수호출
putc (ch, stdout );             // 마크로일수 있다
```

fputc 와 putc 의 파라메터들과 돌림값은 다음과 같다.

파라메터 1: 출력할 문자
파라메터 2: 출력선의 흐름
결과: 조작의 성공 혹은 실패

한 문자입력

```
ch = fgetc(sp);              //함수호출
ch = getc(sp);               //마크로일수 있다
```

fgetc 와 getc 의 파라메터들과 돌림값들은 다음과 같다.

파라메터 1: 문자가 입력되여야 할 흐름

결과 : 문자입력. 파일의 마지막문자가 흐름선상에 나타났을 때 EOF 를 돌려 준다. EOF 는 <stdio>에 정의되여 있다.

흐름닫기

```
fclose (sp);
```

fclose 의 파라메터들과 돌림값은 다음과 같다.

파라메터 1: 닫겨 져야 할 흐름

결과 : 조작의 성공 혹은 실패

파일에서 임의로 위치를 정하기

파일의 시작부터 파일의 위치를 12 바이트 이동하는 실례는 다음과 같다 .

```
fseek(sp,12L, 0);
```

fseek 의 파라메터들과 돌림값들은 다음과 같다.

파라메터 1: 임의위치정하기가 실현되여야 할 흐름

파라메터 2: 이동되여야 할 바이트수인데 이 수자는 long 값이여야 한다.

파라메터 3: 0 파일시작
1 현재위치
2 파일끝

결과 : 조작의 성공 혹은 실패

입출력간략법(shortcuts)

다음의 함수들은 표준입출력함수들의 특수경우이다.

함수	대응되는 함수
printf (args);	fprintf (stdout, args);
putchar (ch);	putc (ch, stdout);
getchar ( );	getc (stdin);

부록 3. 함수

여기에서 보여 주는 루틴(routine)들과 마크로들은 일반적인 C표준서고부분이다. 그러나 대부분의 루틴들은 C++문자렬과 같이 저준위구조들에서 작용한다. 이러한 루틴들을 리용할 때에는 파라메터들이 수행될 함수에 대하여 타당한 자료를 포함하고 있는가를 확인하여 보는것이 좋다. 일부 경우 함수의 정확한 결과는 실현부에 따른다. 실례로 프로그람내부에서 일감조종언어(JCL:Job Control Language) 지령을 실행하는 함수 system을 들수 있다.

주의: 아래에 서술된 루틴들은 제공된 자료에 대해 간단한 오유검사를 진행한다. 대체로 틀리거나 타당치 못한 자료들은 예상할수 없는 결과를 산생시킨다.

문자형과 변환

다음의 마크로들은 문자가 영문자, 수자 등인가를 결정하는데 쓰인다. 넘기는 문자는 부호 없는 문자(unsigned char)로 표시될수 있어야 한다. 이 함수들을 리용하자면 마크로정의가 들어 있는 ctype.h파일이 포함되여야 한다.

include <ctype>

마크로	설 명
int isalnum (int c)	c가 수자나 문자이면 TRUE, 아니면 FALSE를 돌려 준다.
int isalpha (int c)	c가 문자이면 TRUE, 아니면 FALSE를 돌려 준다.
int iscntrl (int c)	c가 조종문자이면 TRUE, 아니면 FALSE를 돌려 준다.
int isdigit (int c)	c가 수자이면 TRUE, 아니면 FALSE를 돌려 준다.
int islower (int c)	c가 소문자이면 TRUE, 아니면 FALSE를 돌려 준다.
int isgraph (int c)	c가 인쇄문자이면 TRUE, 아니면 FALSE를 돌려 준다.
int isprint (int c)	c가 공백을 포함한 인쇄문자이면 TRUE, 아니면 FALSE를 돌려 준다.
int ispunct (int c)	c가 공백을 포함한 구두점문자이면 TRUE, 아니면 FALSE를 돌려 준다.
int isspace (int c)	c가 모든 형식의 공백(실례로 공백건,타브건,행바꾸기건,수직타브 등에 의한 공백)이면 TRUE, 아니면 FALSE를 돌려 준다.
int isupper (int c)	c가 대문자이면 TRUE, 아니면 FALSE를 돌려 준다.
int isxdigit (int c)	c가 16진수이면 TRUE, 아니면 FALSE를 돌려 준다.

다음의 함수들은 대소문자사이의 변환을 진행한다. 이 함수들을 리용하자면 마크로정의가 들어 있는ctype. h 파일이 포함되여야 한다.

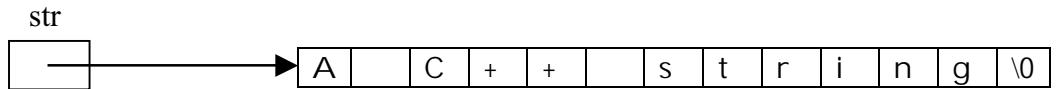
include <ctype>

함수	설 명
int tolower (int c)	대문자를 해당한 소문자로 변환한다.
int toupper (int c)	소문자를 해당한 대문자로 변환한다.

C 형식의 문자렬함수

C 형식의 문자렬은 문자 "\0"으로 끌나는 문자들의 벡토르가 들어 있는 기억구역을 가리키는 문자지적자에 의해 표현된다.

str

A		C	+	+		s	t	r	i	n	g	\0

이 함수들을 리용하기 위하여서는 함수원형정의가 들어 있는 string.h 파일을 포함해야 한다.

include <string>

함 수	설 명
char *strcat (char *d, const char *s)	s문자렬을 d문자렬뒤에 추가한다. 돌림값은 새로운 s문자렬의 지적자이다.
char strcpy (char *d, const char*s)	s문자렬을 "\0"을 포함한 d문자렬에 복사한다. 돌림값은 새로운 s문자렬의 지적자이다.
int strcmp (const char *s1, const char *s2)	문자렬 s1과 s2에 대한 비교를 진행한다. 이때 다음의 옹근수를 돌려 준다. <0 s1이 s2보다 작은 경우. ==0 s1과 s2가 같은 경우. >0 s1이 s2보다 큰 경우.
size_t strlen (const char *s)	s문자렬의 길이를 돌려 준다.
char *strncmp(char *d, const char *s, size_t max)	strcmp와 같은데 단지 max문자크기만큼 비교한다
char *strncpy(char *d, const char *s, size_t max)	만일 s문자렬이 max보다 길지 않다면 max 크기만한 길이의 s를 "\0"을 포함한 d에 복사한다.

주의: 머리부파일 stddef.h 에는 size-t 가 정의되여 있는데 size-t 는 옹근수로 정의되여 있다.

그밖의 문자렬함수들

함 수	설 명
char strchr (const char *s,int c)	s문자렬을 왼쪽에서 오른쪽으로 주사하면서 c문자를 처음 만나는 문자의 지적자를 돌려 준다. c가 s문자렬에 없으면 null을 돌려 준다.
size_t strcspn(const char*s1, const char*s2)	s1 과 s2 를 비교하면서 같지 않은 문자들의 길이를 돌려 준다.
char *strerror(int n)	오유번호 n 에 대한 설명문자렬을 돌려 준다 (보통 입력/출력함수로부터).

함 수	설 명
int stricmp(const char *s1, const char *s2)	strcmp지령과 비슷한데 "ABC"와 "abc"인 경우의 비교결과는 같지 않다고 본다.
char *strlwr (char *s)	s에 있는 대문자를 소문자로 변환한다.
char *strupr(char *s)	s에 있는 소문자를 대문자로 변환한다.
char *strncat(char *d,const char *s, size_t max)	max크기만한 길이의 d를 s에 추가하고 "\0"을 붙인다.
char *strpbrk(const char *s1, const char *s2)	s2와 s1을 비교하면서 처음으로 같은 문자가 나올 때 s1의 지적자를 돌려 주며 그렇지 않으면 null을 돌려 준다.
char *strrchr(const char *s,int c)	strchr와 같은데 주사를 오른쪽에서 왼쪽으로 한다.
size_t strspn (const char *s1, const char *s2)	s1과 s2를 비교하면서 같은 문자들의 길이를 돌려 준다.
char *strtok(char *s1, const char *s2)	s2의 문자들에 의한 구분되는 s1의 토큰들을 돌려 준다. 이 함수를 반복호출하면 s1의 다음번 토큰을 돌려 주며 이때 돌림값이 null값이면 토큰이 더이상 없다는것을 가리킨다.

수학함수

이 함수들을 리용하기 위하여서는 표준함수 math.h를 포함해야 한다. 일부 경우에 함수에 무효한 값이 제공되면 그 결과는 정의되지 않는다.

include <math>

함 수	설 명
double sin (double x)	시누스
double cos (double x)	코시누스
double tan (double x)	탕겐스
double asin(double x)	아크시누스
double acos(double x)	아크코시누스
double atan(double x)	아크탕겐스
double atan2(double x,double y)	x ÷ y 의 아크탕겐스
double sinh(double x)	쌍곡시누스
double cosh(double x)	쌍곡코시누스
double tanh(double x)	쌍곡탕겐스

double exp(double x)	e^x
double log(double x)	자연로그
double log10(double x)	상용로그

함　수	설　명
double pow(double x,double y)	x^y
double sqrt(double x)	루트
double ldexp(double x,int n)	$x.2^n$
double fmod(double x,double y)	x ÷ y 의 소수부값인데 부호는 x 와 같다
double ceil(double x)	x 보다 작지 않은 제일 작은 온근수
double floor(double x)	x 보다 크지 않는 제일 큰 옹근수
double fabs(double x)	x 의 절대값(부수 x 는 정수로 된다)

다음의 함수들은 옹근수의 절대값을 구한다. 이 함수들을 리용하기 위하여서는 표준함수서고 <stdlib.h>를 포함해야 한다.

include<stdlib>

함　수	설　명
int abs(int n)	int n 의 절대값을 돌려 준다.
long labs(long n)	long n 의 절대값을 돌려 준다.

우연수발생

아래에 보여 주는 함수들은 우연수발생에 리용된다. 이 함수들을 리용하기 위하여서는 표준함수서고 stdlib.h 를 포함해야 한다.

include <stdlib>

함　수	설　명
int rand (void)	0 부터 RAND_MAX 범위에 있는 준우연수를 돌려 준다.
void srand (unsigned int n)	준우연수계렬의 초기값을 설정한다. 기정초기값은 1 이다.

문자렬을 수로 변환하기

아래에 보여 주는 함수들은 C++문자렬에 들어 있는 수들을 그 수의 물리적표현으로 변환하는데 리용된다. 이 함수들을 리용하기 위하여서는 표준함수서고 stdlib.h 를 포함해야 한다

include <stdlib>

함　수	설　명
double atof(const char *s)	s 문자렬로 표현된 수를 double 형으로 변환한다.
int atoi (const char *s)	s 문자렬로 표현된 수를 int 형으로 변환한다.
long atol (const char *s)	s 문자렬로 표현된 수를 long 형으로 변환한다.

C++프로그람에서의 탈퇴

다음의 함수는 C++프로그람에서 탈퇴할 때 쓴다. 이 함수들을 리용하기 위하여서는 표준함수서고 stdlib.h를 포함해야 한다.

#include <stdlib>

함 수	설 명
void abort (void)	오유 SIGABT(이상완료)로서 현재실행프로그람이 중지된다.
void exit (const int n)	상태 n 을 가지고 프로그람에서 탈퇴한다. exit(0)은 프로그람의 정상완료이다.
int atexit (void(*fun) (void))	프로그람이 끝나면서 함수 fun 이 호출된다. 함수등록을 할수 없다면 돌림값은 령이 아니다.

프로그람내부에서 지령선의 실행

다음의 함수는 실행하고 있는 C++프로그람내부의 주환경에 의하여 지령을 실행하는데 쓰인다. 이 함수들을 리용하기 위하여서는 표준함수서고 stdlib.h를 포함하여야 한다.

#include <stdlib>

함 수	설 명
int system (const char *s)	문자렬 s에 들어 있는 JCL지령을 실행한다. 돌림값은 그 문자렬을 실행하여 나온 결과이다. 이 함수는 실현부에 의존한다.

오유수정

마크로 assert는 C++프로그람에서 어떤 주장의 타당성을 검사하는데 리용된다. 만일 기호 NDEBUG가 #define NDEBUG로 정의되였다면 마크로는 무시되고 코드는 생성되지 않는다. 이 마크로를 리용하기 위하여서는 마크로정의를 포함하고 있는 assert.h파일을 포함해야 한다.

include <assert>

마크로	설 명
void assert (const int expression)	만일 식이 false 이면 오유통보문 Assertion failed: expression file filename, line number 이 stderr 에 씌여 진다.

함수의 변수들에 대한 접근

만일 함수가 가변개수인수들로 서술되였다면 아래표의 마크로들은 기계적인 방법에 의존하지 않고 이 인수들에 접근할수 있다. 실례로

```
int sum_params(const int last_real_arg, …);
```

이 마크로를 리용하려면 마크로정의가 들어 있는 stdarg.h파일을 포함해야 한다. 그리고 이 마크로를 리용하기전에 변수 p_args를 선언해 주어야 한다.

```
va_list p_args;
```

include<stdarg>

마크로	설 명
va_start(va_list p_args, last_real_arg)	인수목록에서 p_args 를 첫번째 인수로, last_real_arg 를 그 다음인수로 설정한다. 이것은 첫번째 인수의 주소를 가진다.
type va_arg (va_list p_args,type)	변수목록에서 다음인수를 형으로 돌려 준다.
void va_end(va_list p_args)	인수가 처리된 다음 그 처리를 지우기 위해 이 마크로가 반드시 호출되여야 한다.

부록 4. 문자렬클라스

문자렬은 string클라스의 구체례로 실현된다. 이 클라스의 실현은 두개의 본보기함수와 여러개의 기초클라스들을 포함한다. 이 함수를 리용하기 위하여 <string> 파일을 다음과 같이 선언하여야 한다.

#include<string>

클라스 string의 성원들에 대한 요약목록은 아래와 같다. 여기서 함수가 문자렬을 돌려 줄 때 그것은 string 클라스의 구체례이거나 string클라스의 구체례를 가리킨다. 파라메터들의 형은 다음과 같다.

파라메터	형	설 명
pos	type_t	문자렬의 위치. 첫 위치는 0이다.
str	string	문자렬
no	type_t	선택한 문자렬의 개수

클라스 string의 주요메쏘드들은 다음과 같다 .

메쏘드	설 명
< <= != == >= >	두 문자렬을 비교한다.
+	두개의 C++문자렬을 련결하여 돌려 준다.
[i]	객체이름의 i 번째 문자를 돌려 준다.
capacity( )	문자렬용량을 돌려 준다. 그것은 문자렬크기와 같거나 그보다 더 크다.
compare (str)	현재객체와 str를 비교한다.

메쏘드	설 명
c_str ( )	본문에 들어 있는 C++형식의 char*문자렬을 돌려 준다.
find(str)	문자렬에서 처음으로 나타나는 str를 돌려 준다.
get_at (pos)	문자렬에서 pos위치의 문자를 돌려 준다.
insert (pos,str)	문자렬의 pos위치에 str문자렬을 삽입한다.
length ( )	보관된 문자렬에서 문자들의 개수를 돌려 준다.
put_at (pos,ch)	pos위치의 문자를 ch와 교체한다.
remove(pos,no)	pos위치에서 no문자로 시작하는 문자렬을 제거한다.
replace (pos,no,str)	pos위치에서 no문자로 시작하는 문자렬을 str로 교체한다.
substr(pos,no)	pos위치에서 시작하는 새로운 문자렬을 no문자길이만큼 돌려 준다.

주의: 탐색 혹은 추출에서 실패는 문자렬 string::npos를 돌려 주는것에 의해 표시된다.

string 클라스가 std이름령역의 성원이므로 그 클라스의 구체례는 다음과 같이 선언된다.

```
std :: string name;
```

부록 5. 표준서고

표준서고서술에서 다음과 같은 류형의 반복자들이 쓰인다.

반복자	설 명	다음과 같은것에 의해 만들어 진다
IT	입력반복자. 앞방향이동으로 읽기만 한다.	istream_iterator
OI	입력반복자. 앞방향이동으로 쓰기만 한다.	ostream_iterator inserter, front_inserter, back_inserter
FI	앞방향반복자. 앞방향이동으로 읽기, 쓰기한다.	vector deque,list
BDI	쌍방향반복자. 앞뒤방향이동으로 읽기, 쓰기한다.	list, set, multiset, map, multimap
RAI	임의접근반복자. 임의접근으로 읽기, 쓰기한다.	vector, deque

반복자들은 계층적이므로 앞방향반복자는 입출력반복자가 요구되는데서 리용될수 있으며 임의접근반복자는 임의의 다른 반복자가 요구되는데서 리용될수 있다.

초기화알고리듬

알고리듬	설 명
void fill(FI first,FI last,const T& val)	while (first != last) *first++ = val;
void fill_n(OI p,int size,const T&val)	while (size--) construct(&*p++, val)
OI copy(IT first,IT last,OI result)	while (first != last) *result++ = *first++;
OI copy_backwards (IT first, IT last, OI result)	while (first != last) *--result++= --first;
void generate(FI first,FI last,Generator g)	while (first != last) *first++ = g( );
void generate_n(OI p,size,Generator g)	while (size--) *p = g( )
void swap_ranges(FI first,FI last,FI first2)	범위내에서 값교환

탐색알고리듬

알고리듬	설 명
IT find (FI first,FI last, const T& val)	Val이 처음으로 나타나는 요소의 지적자를 돌려 준다.
IT find_if(FI first,FI last, Predicate)	Predicate를 만족하는 첫 요소의 지적자를 돌려 준다.
FI adjacent_find(FI first, FI last [,fun])	어떤 같은 요소의 다음에 있는 첫 요소의 지적자를 돌려 준다.
const T& max (const T& f, const T& s [,compare])	조에서 최대값을 돌려 준다.
const T& min (const T& f, const T& s [,compare])	조에서 최소값을 돌려 준다.
FI max_element(FI first, FI last [,compare])	최대값요소의 지적자를 돌려 준다.
FI min_element(FI first, FI last [,compare])	최소값요소의 지적자를 돌려 준다.
pair<IT,IT> mismatch (IT first1, IT last1, IT first2, IT last2 [,fun])	서로 맞추어 지지 않는 첫 요소의 지적자쌍을 돌려 준다.

변환알고리듬

알고리듬	설 명
void reverse(BDI first,BDI last)	요소순서를 반전한다.
void replace (FI first,FI last, const T& old, const T& new)	낡은 요소값을 새로운 요소값으로 교체한다.
void replace_if(FI first,FI last, Predicate,const T& new)	predicate가 true인 새 값으로 요소들을 교체한다.

알고리듬	설 명
void replace_copy(IT first,IT last,OT result, Const T& old,const T& new)	새로운 복사를 만든다.
void replace_copy(IT first,IT last,OT result, Predicate,const T& new)	새로운 복사를 만든다.
void rotate(FT first,FT mid,FT last)	가운데를 중심으로 서로 요소들을 바꾼다.
BDI partition (BDI first, BDI mid, BDI last, Predicate)	요소를 분리하여 Predicate를 만족하는것들은 앞으로 이동된다.
BDI stable_partition(BDI first, BDI mid, BDI last, Predicate)	우와 같으나 본래순서는 유지한다.

이동알고리듬

알고리듬	설 명
FI remove (FI first,FI last,const T& val)	val요소들을 작아 지는 차례로 다시 쓴다. 새로 만들어 진 마지막요소에 대한 지적자를 돌려 준다.
FI remove(FI first,FI last,Predicate)	우와 같으나 predicate에 맞는 요소들만 움직인다.
FI remove_copy(IT first,IT last, OI result, Const T& val)	remove와 같지만 복사본을 만든다.
FI unique (FI first,FI last [, Binarypredicate])	반복요소를 없애며 새로 만들어 진 마지막요소의 지적자를 돌려 준다.

스칼라발생알고리듬

알고리듬	설 명
void count (IT first, IT last, const T& val,Size& count)	val과 같은 요소가 나타날 때마다 계수를 증가시킨다.
void count_if(IT first, IT last, Predicate f,Size& count)	Predicate와 같은 요소가 나타날 때마다 계수를 증가시킨다.
containerType accumulate(IT first,IT last, ContainerType initial [, BinaryFun])	매 요소에 대하여 BinaryFun(기정값 +)을 수행한 결과를 돌려 준다.

일반알고리듬

알고리듬	설 명
Function for_each(IT first, IT last, Function);	매 요소에 Function을 적용한다.

정렬알고리듬

알고리듬	설 명
void sort (RAI first, RAI last [,compare])	정렬
void stable_sort(RAI first, RAI last [,compare])	같은 요소의 순위를 유지하면서 정렬
void partial_sort(RAI f, RAI m, RAI l [,compare])	f부터 m까지 정렬시킨다.
void partial_sort_copy(IT f1, IT l1, RAI f, RAI l [,compare])	

탐색알고리듬

알고리듬	설 명
void nth_element(RAI first, RAI nth, RAI last [, compare])	n번째 반복자를 경계로 하여 순서대로 부분분류한다.
bool binary_search(FI first, FI last, const T& value[, compare])	값이 존재한다면 true를 돌려 준다(집합은 순서대로 있어야 한다).
FI lower_bound(FI first, FI last, const T& value[,compare])	순서분류한 다음 반복자의 첫 위치를 돌려 준다.
FI upper_bound(FI first, FI last, const T& value [, compare])	순서분류한 다음 반복자의 마감위치를 돌려 준다.
Pair<FI, FI> equal_range(FI first, FI last const T& value [, compare])	반복자의 첫 위치와 마감위치를 한조로 하여 돌려 준다.

합동알고리듬

알고리듬	설 명
OI merge(IT first1, IT last1, IT first2, IT last2, OI result [, compare])	범위1 과 범위2를 result에 합동한다. 이때 두개 범위가 같다면 범위1이 먼저 놓인다.

모임연산알고리듬

알고리듬	설 명
OI set_union(IT first1, IT last1, IT first2, IT last2, OI result [, compare])	합모임
OI set_itersection(IT first1,IT last1,IT first2, IT last2, OI result [,compare])	사귐모임
OI set_difference(IT first1, IT last1, Itfirst2, IT last2, OI result [, compare])	차모임
OI set_symmetric_difference(IT first1, IT last1, IT first2, IT last2, OI result [,compare])	대칭적인 차모임
bool includes (IT first1,IT last1, IT first2, IT last2)	모임1이 모임2의 부분이라면 true를 돌려 준다.

더미령역조작알고리듬

알고리듬	설　명
void make_heap(RAI first,RAI last, [,compare])	더미령역에 우연수렬을 만든다.
void push_heap(RAI first,RAI last, [,compare])	마지막에 한개 요소를 추가한후 더미령역이 다시 보관된다.
void pop_heap(RAI first,RAI last,[,compare])	첫 요소와 마지막요소를 바꾸고 마지막요소를 뺀 더미령역속성을 다시 보관한다.
void sort_heap(RAI first,RAI last,[,compare])	순서화된 집합으로 변환한다(더미령역속성은 유지한다).

함수객체

함수	연산	함수	연산
더하기	x+y	덜기	x-y
곱하기	x*y	나누기	x/y
모듈러연산	x%y	부정	-x
같기	x==y	같지않기	x!=y
크기	x>y	작기	x<y
크거나 같기	x>=y	작거나 같기	x<=y
론리적	x&&y	론리합	x||y
론리부정	!x		

사용실례는 다음과 같다.

```
std ::cout<<”2+3=”<<plus<int> ( ) (2,3)<<”\n”;
```

함수적응자

함　수	설　명
not1(unary_function)	unary_function과 같이 함수를 돌려 준다. 본래 함수와 같지 않은것만 돌려 준다.
not2(binary_function)	binary_fnction과 같이 함수를 돌려 준다. 본래 함수와 같지 않은것만 돌려 준다.
bindlst(binary_function,arg1)	2원함수 Tf(T2x,T2y)를 Tf(arg1,y)를 실현하는 1원함수 Tf(T2y)로 변환한다.
bind2nd(binary_function,arg2)	2원함수 Tf(T2x,T2y)를 Tf(x,arg2)를 실현하는 1원함수 Tf(T2,x)로 변환한다.

용기클라스

표준서고에는 다음의 용기(container)들이 들어 있다.

용 기	설 명	머리부파일
vector	임의접근을 진행하며 벡토르마감에 새 요소 삽입	<vector>
list	목록의 임의의 장소에 요소의 삽입과 삭제	<list>
deque	구조체의 앞뒤에서 삽입과 삭제	<deque>
set	요소들을 순서대로 놓고 용기에 대한 포함, 삽입, 삭제를 검사	<set>
multiset	2중값을 가진 모임	<set>
map	열쇠에 의한 값접근, 효률적인 삽입과 삭제	<map>
multimap	2중열쇠에 의한 도표(map)	<map>
stack	앞으로부터만 삽입과 삭제	<stack>
queue	앞에서 항목의 삭제와 뒤에서만 항목의 삽입	<queue>
Priority Queue	제일 큰 값에 대한 접근과 삭제	<queue>
bitset	요소(비트)들의 순서유지, 포함과 삽입, 삭제를 위한 검사	<bitset>

용기요구

식	돌림값
c::value_type	용기에 기억되는 형
c::reference	용기에 기억되는 항목의 왼쪽값(주소)
c::const_reference	용기에 기억되는 형의 상수오른쪽
c::iterator	반복자형
c::const_iterator	고정반복자형
c::difference_type	
c::size_type	용기의 항목수를 표시할수 있는 형

식	돌림값
co.begin( )	집합의 시작에 있는 co의 반복자
co.end( )	집합의 마감에 있는 co의 반복자
ca == cb	용기 ca와 cb의 비교
co.size( )	용기 co의 요소수
co.max_size( )	용기의 최대크기
co.empty( )	용기가 비여 있는 경우
ca < cb	

반전가능한 용기요구

이것들은 순서를 반대로 바꿀수 있는 용기를 위한것이다(실례로 vector, list, set, multiset, map, multimap이다).

식	돌림값
c::reverse_iterator	반대방향으로 진행할수 있는 반복자
c::const_reverse_iterator	반대방향으로 진행할수 있는 고정반복자
co.rbegin()	집합의 끝에 있는 co의 역반복자
co.rend	집합의 시작에 있는 co의 역반복자

순차용기요구

이것들은 항목의 삽입과 삭제를 진행하는 용기를 위한것이다(실례로 vector, list, set, multiset, map, multimap이다).

식	실 현
co.insert(p,item)	용기 co에 있는 p앞에 item을 삽입한다.
co.insert(p,n,item)	n항목들을 삽입한다.
co.erase(p)	p로 지적되는 요소를 지운다.
co.erase(p,q)	지적자범위 p에서 q까지 요소들을 지운다.

선택순차용기요구

이것들은 용기우에서의 연산에 걸린 시간이 고정되여 있을 때만 제공된다(실례로 vector, list, set, multiset이다).

식	실 현
co.front( )	첫 요소를 돌려 준다.
co.back( )	마지막요소를 돌려 준다.
co.push_front(item)	앞에 새 항목을 삽입한다.
co.push_back(item)	뒤에 새 항목을 삽입한다.
co.pop_front( )	앞요소를 삭제한다.
co.pop_back( )	마지막요소를 삭제한다.
co[i]	용기의 i번째 요소를 돌려 준다 (vector용기에서만).

결합순차용기요구조건

식	실 현
c::key_type	열쇠
c::key_compare	순서대로 열쇠에 의한 비교
c::valu_compare	값에 의한 비교

부록 6. 연산자우선순위

결합	C++에서 연산자들 우선순위가 높은것으로부터 낮은것에로	주의
왼쪽에서 오른쪽으로	( ) [] -> . ::	
오른쪽에서 왼쪽으로	! ~ ++ -- +- * & (type) sizleof new delete	단항연산자
왼쪽에서 오른쪽으로	.* ->*	
왼쪽에서 오른쪽으로	* / %	
왼쪽에서 오른쪽으로	+ -	
왼쪽에서 오른쪽으로	<< >>	
왼쪽에서 오른쪽으로	< <= >= >	
왼쪽에서 오른쪽으로	== !=	
왼쪽에서 오른쪽으로	&	
왼쪽에서 오른쪽으로	^	
왼쪽에서 오른쪽으로	|	
왼쪽에서 오른쪽으로	&&	
왼쪽에서 오른쪽으로	||	
오른쪽에서 왼쪽으로	? :	조건연산자
오른쪽에서 왼쪽으로	= *= /= etc.	
왼쪽에서 오른쪽으로	,	반점연산자

부록 7. 확장문자렬

조종문자들을 하나의 문자렬이나 문자상수로 표시하기 위하여 쓰이는 확장문자렬들은 다음과 같다.

\ "	겹인용부호	\ddd	8진값으로 된 문자
\ '	외인용부호	\f	페지넘기기
\0	문자렬끝표식자(\뒤에 수자 0이 붙는다.)	\n	행바꾸기
\?	물음표	\r	행앞머리로 가기(return)
\\	\문자	\t	타브
\a	종	\v	수직타브
\b	후진시키기(backspace)	\xddd	16진값으로 된 문자
\c	자리복귀(carrage return)		

따라서

```
std::cout << “ \ ” A string \ “ \n \t with embedded escape sequences ”
```

은 다음의것을 인쇄한다.

```
“A string”
  with embedded escape sequences
```

부록 8. 기본형

형	줄임말	길이(바이트)	최소값	최대값
char			0or-127	255 /127
signed char		= char	-127	+127
unsigned char		= char	0	255
w_char				
bool				
int		>= short	-32767	32767
unsigned int		= int	0	65535
short int	short	>= char	-32767	32767
unsigned short int	unsigned short	= short int	0	65535
long int	long	>= int	-2147483647	2147483647
unsigned long int	unsigned long	= long int	0	4294967295
float			소수부 6자리	10± 37
double		>= float	소수부 10자리	10± 37
long double		>= double	소수부 10자리	10± 37

형 형의 이름이다.
줄임말 이름에 대해 허용가능한 줄임말이다.
길이 그 형과 기억바이트사이의 관계이다.
최소값 표시할수 있는 최소값이다.
최대값 표시할수 있는 최대값이다.
Δ 값의 최소범위.

주의: 콤퓨터에 따라 C++실행시 char는 signed 혹은 unsigned일수 있다.
signed는 옹근수형앞붙이로 사용할수 있다.

부록 9. C++에서의 직접값

C++에서 직접값(literal)은 형을 가지는데 그것은 처리되는 방법에 따라 달라 진다.

직접값	실례	형	설명문
문자	'A'	char	
10진수	123456	int long int	int 등의 실현크기에 따르는 형
부호 없는 수	1234U	unsigned int unsigned long int	U 혹은 u는 뒤붙이
8진수	01234	int long int	0은 8진수를 의미
16진수	0xFACE	int long int	0x 혹은 0X는 16진수를 의미
큰 수	1L	long	L 혹은 l은 뒤붙이
부호 없는 큰 수	123456UL	unsigned long	뒤붙이는 UL,ul 혹은 어떤 결합자이다
실수	1.23F	float	F 혹은 f는 뒤붙이
큰 실수	1.23	double	
매우 큰 실수	1.23L	long double	L 혹은 l은 뒤붙이

주의: 옹근수는 int, unsigned int, long int, unsigned long int에서 제일 작은 표시단위의 형을 가진다. 직접수의 형은 실행시 기본형의 크기에 따라 다를수 있다.
직접수의 실수(실례로 1.23)는 기정으로 double형이다.
옹근수앞에 있는 0은 수자가 8진수라는것을 의미한다.

부록 10. C++에서의 예약어들

asm	auto	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	operator	private	protected
public	register	reinterpret_cast	return	short
signed	sizeof	static	static_cast	struct
switch	template	this	throw	true
try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile
wchar_t	while			

주의: asm은 실행프로그람의 정보를 아쎔블러에 넘기는 예약어이다.

연산자

^	~	!	!=	%	%=
&	&&	&=	( )	*	*=
+	++	+=	'	-	-=
->	->*	.	.*	/	/=
:?	<	<<	<<=	<=	=
^=	==	>	>=	>>	>>=
delete	new	sizeof	[]	|	||

연산자들의 다른 표현

연산자	다른 표현	연산자	다른 표현
&	bitand	&&	and
|	bitor	||	or
^	xor	~	comp1
&=	and_eq	|=	or_eq
^=	xor_eq	!	not
!=	not_eq		

마크로처리기에서 사용되는 예약어들과 토큰들

#	##	define	elif	else
endif	error	ifdef	ifndef	include
defined				

부록 11. C++프로그람에로의 자료넘기기

일부 콤퓨터체계에서는 정보를 외부환경에서 C++프로그람(호출된)에로 넘길수 있다. 이것은 일반적으로 2진프로그람이름뒤에 문자렬토큰(정보)들을 차례로 붙여쓰는 방법으로 실현할수 있다. 실례로 2진프로그람이름이 a.out일 때 이 프로그람을 불러 내고 여기에 통보문 《hello world》를 넘겨 주는 형식은 다음과 같다.

```
a.out hello world
```

이때 C++프로그람은 2개의 파라메터 argc와 argv를 가지고 호출된 함수본체를 가진다. 그 파라메터는 다음과 같은 의미를 가진다.

argc 넘겨 받는 파라메터수를 계수

argv 통보문의 개별적인 토큰들이 들어 있는 문자렬에 대한 지적자들로 된 배렬

주의: argc의 값은 최소한 1이다. 왜냐하면 첫번째 문자렬에는 불러 들인 프로그람의 이름이 들어 가기때문이다.
문자렬이 프로그람에 넘어 오지 않았다면 argc는 1로 되며 argv에는 불러 들인 프로그람이름을 가리키는 지적자가 들어 있는 1개 요소로 된 벡토르로 된다.

통보문의 토큰들을 처리하기 위한 프로그람을 작성할수 있다. 아래의 프로그람은 앞부분(프로그람이름)을 제외한 모든 토큰들을 화면에 표시한다.

```
int main (const int argc,  const  char* const  argv[ ])
{
   for(int   i =1, i < argc;  i++)
   {
       std::cout  <<  argv[ i ]  <<   ( i != argc ?  “   ” : “ ” );
   }
   std::cout << “ \n ” ;
   return   0;
}
```

이 프로그람은 UNIX체계에서 다음과 같이 실행될수 있다.

```
% a.out This should say  Hello  world
```

이때 결과는 다음과 같다.

```
This  should  say   Hello  world
```

주의: UNIX체계에서 이와 비슷한 프로그람은 echo이다.

부록 12. C++프로그람에서 C함수에 대한 접근

C++는 safe형련결을 리용한다. 이 처리는 자기형의 설명이나 해당한 파라메터들의 설명을 포함하기 위해 C++외부함수나 외부변수이름들을 분할한다. 련결편집기는 그때 류사하게 분할된 이름들을 결합하기만 한다. 다시 말하여 C++가 아닌 다른 언어로 씌여 진 코드는 련결하기 힘들다. 왜냐하면 그 이름들이 호환방식으로 분할되지 않기때문이다.

이 이름분할은 함수나 변수앞에 extern”C”를 붙여 막을수 있다. 실례로 C함수 written_in_c

```
float written_in_c(float number)
{
/*  이미 정의된 함수인데 C++콤파일러로 다시 콤파일할수 없다*/
}
```

는 C++프로그람에서 지정된 다음과 같은 원형을 가진다.

```
extern “C” float written _in_c( float );
```

이것은 련결편집기가 C프로그람에 written_in_c이름을 결합할수 있게 하면서 그 이름을 분할하는것을 중지한다.

C머리부파일은 C++프로그람에서 다음과 같이 사용될수 있다.

```
extern “C ” {
# include “C_header_file.h ”
}
```

이 코드는 함수원형이 머리부파일로 정의되여 있는 함수들에 대한 모든 호출에 대하여 이름분할을 중지한다.

주의: C프로그람들도 선택적으로 콤파일할수 있는 대부분의 C++콤파일체계들에서는 C 프로그람이 C++프로그람에 포함될 때 C머리부파일들에 extern“C”를 포함하는 조건적콤파일을 사용한다.

부록 13. 코드의 호환성

이 책에서 보여 주는 프로그람들은 Borland C++ Builder Version3으로 콤파일하였다.

콤파일러	문제점	해결책
ANSI C++ Compiler	없다.	
Borland C++ Version 5.02	클라스에서 상수성원들은 초기화되지 않는다.	26장 유산콤파일러를 보시오.
Borland C++ Version 4.5	for순환에서 시작값명령문의 유효범위. namespace지령은 실행되지 않는다. mutable수식자는 실행되지 않는다. explict수식자는 실행되지 않는다. 클라스에서 상수성원은 초기화되지 않는다.	26장 유산콤파일러를 보시오.
Borland C++ Version3.0	bool형은 실행되지 않는다. for순환에서 시작값명령문의 유효범위. 례외기구는 실행되지 않는다. namespace지령은 실행되지 않는다. mutable수식자는 실행되지 않는다. explict수식자는 실행되지 않는다. 클라스에서 상수성원은 초기화되지 않는다.	26장 유산콤파일러를 보시오.
GCC egcs-1.0.2	namespace지령은 전혀 실행되지 않는다. mutable수식자는 실행되지 않는다. explict수식자는 실행되지 않는다.	26장 유산콤파일러를 보시오.

참고문헌

Margaret A Ellis & Bjarne Stroustrup,
The Annotated C++ Reference Manual. Addison Wesley, 1990.

Bjarne Stroustrup,
The C++ Programming Language, 3nd Edition. Addison Wesley, 1997.

Borland C++ Builder Manuals
(Programmer’s guide, Library reference, and Tools and utilities guide.)
Borland International, 1998.

X3J16/95-0087 WG21/NO687
Working Paper for Draft Proposed International Standard for Information Systems
Programming Language C++. December 1996.

색　인